BENCHMARKING FRAMEWORK FOR SOFTWARE WATERMARKING

BY

MOHANNAD AHMAD ABDULAZIZ AL-DHARRAB

A Thesis Presented to the

DEANSHIP OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

INFORMATION AND COMPUTER SCIENCE

JUNE 2005

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS
DHAHRAN 31261, SAUDI ARABIA

DEANSHIP OF GRADUATE STUDIES

This thesis, written by

Mohannad Ahmad Abdulaziz Al-Dharrab

under the direction of his thesis advisor and approved by his Thesis Committee, has been presented to and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

Thesis Committee:

Dr. Muhammad Alsuwaiyel (Advisor)

Dr. Abdallah Al-Sukairi (Member)

Dr. Sahalu Junaidu (Member)

DR. Kanaan Faisal

Dr. Kanaan Faisal
(Department Chairman)

Dr. Mohammad Al-Ohali
(Dean of Graduate Studies)

5/12/05

Date
14-6-2005

DEDICATION

This thesis is dedicated to my parents

for the love and support I received all the way in my studies.

To my mother for here sacrifices for bringing the happiness into my life.

To my father who has been a great source of motivation and inspiration.

ACKNOWLEDGEMENTS

First and foremost, all praise to the Almighty Allah for His countless blessings.

Acknowledgement is due to King Fahd University of Petroleum & Minerals and the College of Computer Science & Engineering for all the support to this work.

My sincere appreciation goes to my advisor, Dr. Muhammad Alsuwaiyel for his incomparable guidance, support, and encouragement given to me throughout the course of this work.

Many thanks to my committee members, Dr. Abdallah Al-Sukairi and Dr. Sahalu Junaidu for all the help and support I received.

I offer my deepest gratitude and appreciation to Dr. Lahouri Gouti for his unstinting commitment and generous support to help me through this work to its final completion, also for his insightful comments and wise guidance during my thesis progress.

Finally, acknowledgments are extended to my parents, family members, and sincere friends for their continuous prayers, encouragement, and moral support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

THESIS ABSTRACT

Name: Mohannad Ahmad AbdulAziz Al-Dharrab

Title: Benchmarking Framework for Software Watermarking

Major Field: Information & Computer Science

Date of Degree: June 2005

Software watermarking is one of the most important methods for protecting copyrights and authenticating ownership; and hence preventing software piracy. It received more attention recently and it is expected to even get more interest. However, in all the work done in this area, none of them have used a specific or a particular evaluation measures to criticize the proposed software watermarking technique.

In this thesis, several issues related to software watermarking were studied and a survey was conducted on the current and promising new techniques designed to reliably preserve and protect software programs. Different software watermarking techniques and attacks to those techniques were classified and evaluated. A promising benchmarking framework for software watermarking techniques was proposed, based on the results of conducted experimentations, which allows for measuring the efficiency of current and possibly future proposed software watermarking schemes. This will allow comparing different watermarking techniques and will lead to speeding up the potential research work in this area.

ملخص الرسالــة

الاســــــــــم: مهند بن أحمد بن عبدالعزيز الضراب

عنوان الرسالة: مقاييس وتقييم تقنية البصمة المائية في البرمجيات

(Benchmarking framework for software watermarking)

التخصـــــــص: علوم الحاسب الآلي والمعلومات

تاريخ الشهـادة: ربيع الآخر 1426هـ

تعد البصمة المائية في البرمجيات أحد أهم الطرق لحماية وتوثيق حقوق الملكية للبرامج، وبالتالي منع قرصنة البرمجيات. ولقد حظيت البصمة المائية وتأثيرها في البرمجيات على اهتمام الباحثين مؤخراً، ويتوقع لها المزيد من الاهتمام في المستقبل القريب. وعلى الرغم من وجود عدة أبحاث في هذا المجال، إلا أن أياً منها لم يتطرق إلى استخدام طريقة معينة للقياس أو التقييم أو حتى للحكم على التقنية المقترحة لاستخدام تقنية البصمة المائية في البرمجيات.

في هذه الأطروحة تمت دراسة العديد من المواضيع ذات العلاقة بالبصمة المائية في البرمجيات كما درست تقنيات مختلفة ومعتمدة صممت لحفظ وحماية البرامج. بالإضافة إلى تصنيف وتقييم العديد من التقنيات المختلفة لاستخدام أو مهاجمة البصمة المائية في البرمجيات. وكنتيجة للدراسة التحليلية وبناءاً على نتائج التجارب والاختبارات التي أجريت في هذا البحث، فقد تم وضع طريقة مبتكرة وضوابط لاختبار وتقييم التقنيات المختلفة لوضع البصمة المائية في البرمجيات، وبالتالي إمكانية قياس فعالية وكفاءة تقنيات البصمة المائية في البرمجيات الحالية أو المستقبلية. وستساعد هذه الطريقة على إيجاد آلية واضحة ومحددة لمقارنة تقنيات مختلفة للبصمة المائية في البرمجيات، وبالتالي الإسهام في تطوير وتسريع الأبحاث المستقبلية في هذا المجال.

CHAPTER 1

INTRODUCTION

Software piracy became a major problem with the fast and vast growth in the use of the internet. Moreover, the new computer technologies such as the ability to copy CDs fast and easily aided in increasing software piracy. In fact, many people consider software too expensive to buy and they lack the respect and enforcement of intellectual properties laws. Specialists believe that there is no technique that can prevent all kinds of software piracy though the effort on preventing software piracy is continuing, and hence, the goal is to raise the cost for software pirates.

According to the Business Software Alliance organization (BSA), software piracy caused a loss of nearly $29 billion in year 2003 only, which is 36% of software installed on computers worldwide in the same year. The study, conducted for the first time by global technology research firm International Data Corporation (IDC), incorporated major software market segments including operating systems, consumer software and local market software. The study found that while $80 billion in software was installed on computers worldwide in 2003, only $51 billion was legally purchased [11]. See figure1 for more details.

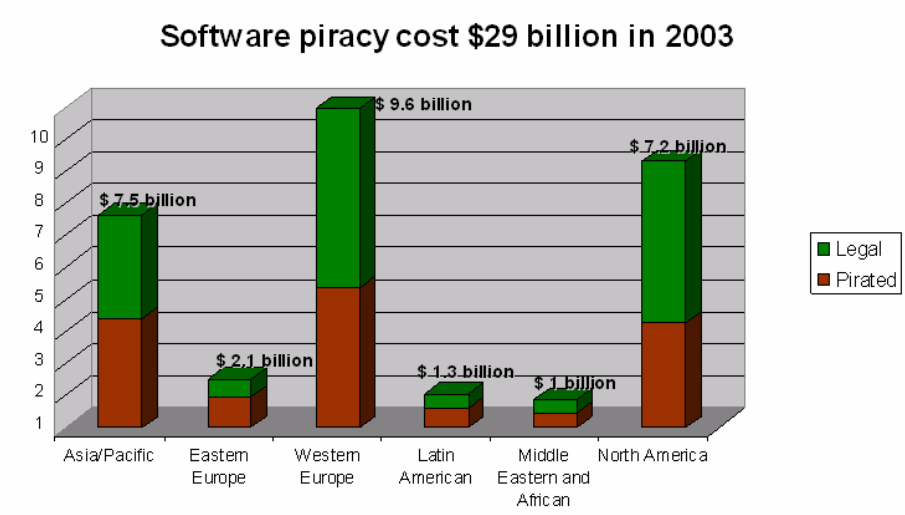

Figure 1: Software piracy cost in the world-wide in year 2003

In 2004, the losses due to software piracy increased from $29 billion to $33 billion though the percentage of pirated software installed dropped one percent from 36% to 35%. The legally purchased software cost more than $59 billion compared with $51 billion and total cost of software installed was over $90 billion compared to $80 billion in 2003. According to BSA president and CEO Robert Holleyman: “Worldwide, one out of every three copies of software in use today has been obtained illegally. These losses have a profound economic impact in countries around the world. Every copy of software used without proper licensing costs tax revenue, jobs, and growth opportunities for burgeoning software markets." [21]

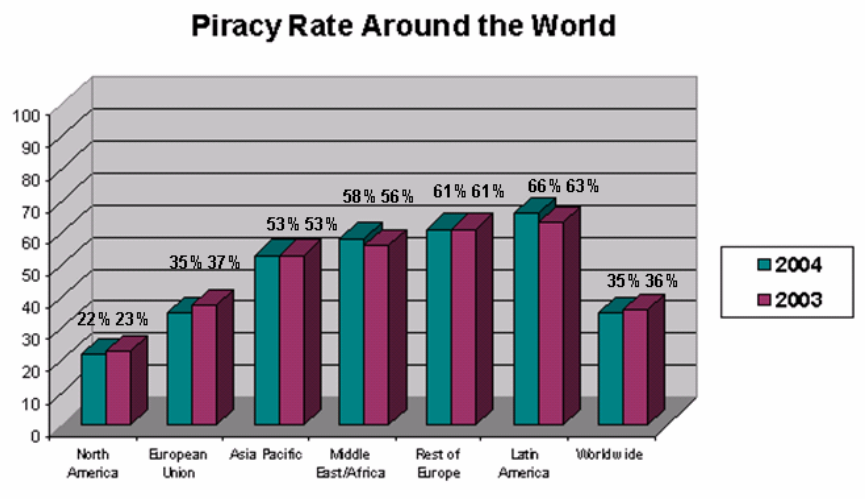

Figure 2: Average Software piracy rates based on regions of the world

1.1 Statement of the Problem

Software watermarking is one of the most important methods in software ownership authentication. In software watermarking, a secret message including the digital rights information is embedded in the software. This secret message can be retrieved later when authentication is required. In this thesis, a survey was conducted on the current and promising new techniques designed to reliably preserve and protect software programs. Different software watermarking techniques and attacks to those techniques were classified and evaluated. The objective was to achieve high embedding capability while maintaining high protection level with an acceptable execution cost for a watermarked software program.

Current stat of art of research on software watermarking has no framework defined for evaluating and standardizing different software watermarking algorithms. The main

contributing of this thesis is to propose a promising benchmark for software watermarking techniques which allows for measuring the efficiency of current and possibly future proposed software watermarking schemes. This will allow comparing different watermarking techniques and will lead to speeding up the potential research work in this area.

1.2 Justification for and Significance of the Study

"Software watermarking is an area that has received very little attention. This is unfortunate since software piracy is rampant. A sizeable fraction, estimated at 39% with a valuation of 13 billion dollars, of business application software is installed annually without a license [BSA2003, Malhotra94]*."* [4].

As mentioned above, research on this area is still considered new and the work done in software watermarking is not much. The area of software watermarking also lacks standardization and efforts done in software watermarking patents are yet limited. The problem at hand is reminiscent of multimedia watermarking where research community did not perceive its importance instantaneously as it was with Cox's paper. As shown in Figure 3, we illustrate the impact of Cox's work on subsequent research in multimedia watermarking. This shows that it has taken four years for the concept to sell itself. On the other hand, Figure 4 illustrates the early life of software watermarking by showing the potentiality and modernity in subsequent research for the work of Collberg and

Thomborson in one of their initial papers where they addressed the problems of software watermarking.

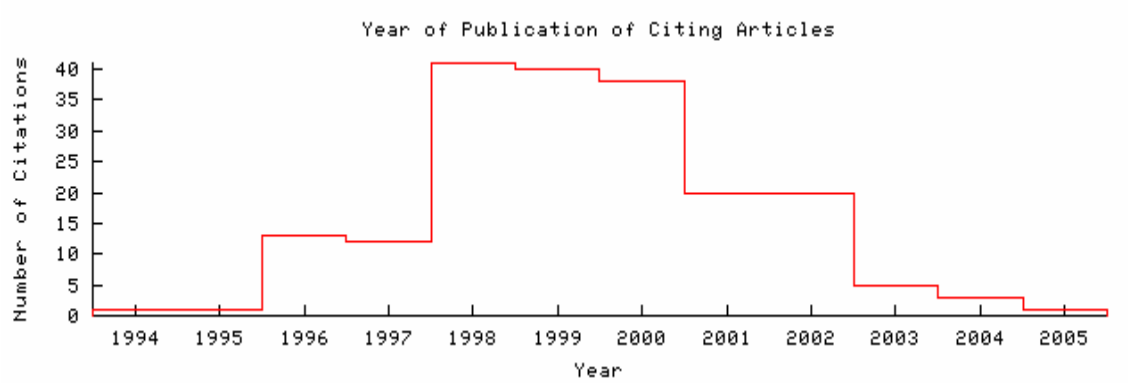

Figure 3: The graph only includes citing articles where the year of publication is known for the paper by Ingemar J. Cox, Joe Kilian, Tom Leighton, Talal Shamoon. "Secure Spread Spectrum Watermarking for Multimedia" (1995) [from CiteSeer.IST].

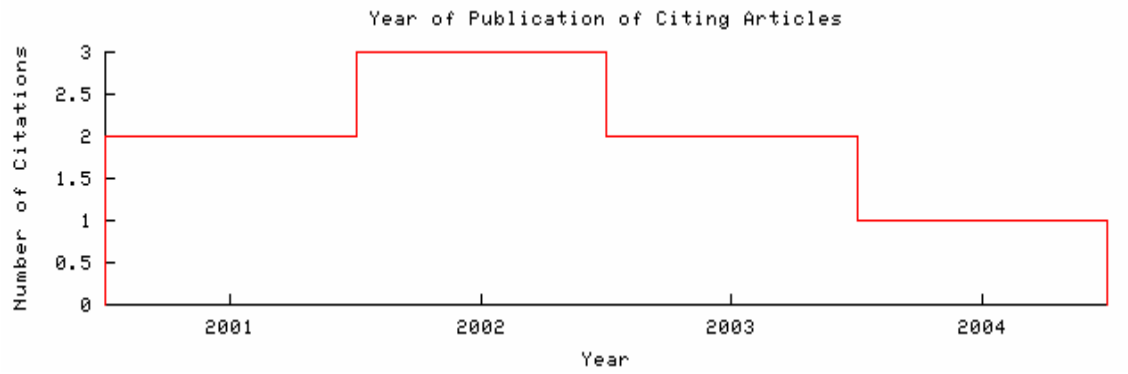

Figure 4: The graph only includes citing articles where the year of publication is known for the paper by Christian Collberg, Clark Thomborson. "On the Limits of Software Watermarking" (1998) [from CiteSeer.IST]

1.3 Thesis Organization

The rest of the thesis is organized as follows. Chapters 2 discuss the importance of software copyrights and ownership authentication. In Chapter 3, different security techniques such as hardware and software techniques are mentioned. The following chapter explains the reason of studying watermarking with java in this thesis. Next, a comparative study on different watermarking algorithms, including static and dynamic watermarks, is presented. Also, several possible attacks on software watermarks are analyzed. The major contributions in this thesis are shown in chapters 6 and 7. Chapter 6 is about an overview of the proposed work. First, the need for evaluation benchmarking and some reminiscent unrelated work is presented. Then, our study approach is enlightened. After that, the proposed benchmarking factors are explained in details. In chapter 7, experimentation results, analysis, and observation are presented. Contributions and results of this thesis are summarized in chapter 7 as well. In the last section, the limitations of the work done and future work are addressed.

CHAPTER 2

SOFTWARE COPYRIGHTS AND OWNERSHIP AUTHENTICATION

2.1 General Background on Software protection

No existing protective measure succeeded in preventing software piracy. Although most software piracy cases can be discovered, following every single case is not feasible and considered costly. In addition, it could take quite some time which might also negatively lead to sliming down the reputation of the software company [1].

In most existing software protection schemes, the nature of defense is by building static techniques in the distributed software. However, breaking the protection once will mean that the software is no more protected. Therefore, no existing technique is perfectly immune to protection attacks [1].

Only few patents exist in computer software protection area. The first one was published in 1994 and claims a method to produce copies of a master file. The basic idea is to put a predetermined block of data within a master copy of a software file. This idea is most

suitable for software distributed over the internet [10]. Another patent claims a method for disabling software copies that are not authorized. It depends on having a set of numbers defined based on an unusual mathematical property [19].

In 1996, a new patent was published which provides a method for generating and auditing a unique signature for executable software. It helps in identifying authorized and non authorized copies [9]. A digital watermarking method became a patent in 1998 aimed to make copy protection of computer software. It encodes and protects computer code copyrights by encoding the code with a digital watermark [13]. This patent is considered the beginning of having watermarks in software programs.

2.2 Causes for Software Piracy

Software creators should think of there software legal users, illegal users, and attackers. In fact, security and copyright issues should be thought of during the software development life cycle as a whole.

Nowadays, software systems are not created by only one single vendor. Instead, multiple parts or subsystems can be assembled together to make the complete software system. This approach brought what is known as commercial off the shelf (COTS) to the software industry. Benefits of having COTS, like saving money, time, and effort, increased the need of such COTS subsystems and their vendors. As a result, those COTS vendors are concerning protecting copyrights and preventing piracy to their software, and hence, they

sell their software products as binary codes or black box components, without giving the design and source code [18].

A driving factor for software piracy is that the cost of producing first copy is very high while producing the following copies is very cheap. Therefore, it's worth adding more value on the marginal cost of producing subsequent copies by making every copy somehow unique. This could be done by having different license number, license file, activation code, decryption key, or fingerprint. However, those unique identifications are not part of the original program and they were added just for protecting the copyrights. This is why they are easy to remove [1].

The fact that some software products are very expensive, almost with the same cost of personal computer, has increased the piracy activities and encouraged individuals to search for illegal low-priced copied software. Software piracy became the business for some individuals and organizations in countries where copyright laws are not seriously enforced. In fact, some illicit organizations are distributing and exporting millions of pirated copies. Resources showed that around 20 billion US dollars are lost yearly only because of software piracy [18].

2.3 Research on Watermarking

Watermarking is defined in the Oxford English Dictionary as follows: “a distinguishing mark or device impressed in the substance of a sheet of paper during manufacture, usually barely noticeable except when the sheet is held against strong light” [14].

The idea of watermarking first appeared hundreds of years ago. Watermarking technology was used to mark information authenticity by many different means. Watermarking technology has been used in computer as well. Most of the work on computer watermarking technology was for embedding a watermark into images, audio, and video files. In the last few years, few research, papers, and patents were published and concerned applying watermarking embedding into software programs.

Media watermarking research is a very active area and digital image watermarking became an interesting protection measure and got the attention of many researchers since the early 1990s [14]. However, software watermarking is a more recent area of interest with a little amount of published work. Research on software watermarking is very few and just started in the last few years although software piracy cost is approximately 20 billion dollars every year.

The issue of software piracy is of concerns of both: software industry and academic community. The reason behind this interest in software piracy is the large economical factors and big losses only because of piracy [1].

As mentioned earlier, multimedia watermarking is a reminiscent for software watermarking. Watermarking is being explored more in the watermarking section of Chapter 3, about different security techniques. In chapter 5 of this thesis, more details about different types of research that was done on software watermarking area.

CHAPTER 3

DIFFERENT SECURITY TECHNIQUES

Enforcing law measures is very important to prevent software piracy and protect copyrights. Having the law by itself is not sufficient and there must be technical measures to be implemented. These technical measures should be reliable enough to prevent piracy and practical enough not to affect the software performance and economic value as well [18].

The main objective of all the protection schemes is to raise the cost for pirates to break the protection approaches [18]. Thus, the higher the cost for the pirates to break the software security, the higher the protection level of the application. This fact leaded to the existence of many software security techniques with varying nature.

To prevent piracy attacks, several approaches could be done. Examples include: creating a list of certified customers, embedding the software into the hardware, and configuring the software to automatically send the computer serial number when connected to the internet. These approaches are so difficult to apply, especially with the improvement in the internet, disassemblers, and de-compiler programs. The limitations of approaches above have increased the research interest on another software protection technique

called software watermarking. The basic idea of software watermarking is to embed a secret message (number) into the program such that when extracted, the owner proves his\here ownership of the software [15].

Many media watermarking algorithms were developed and obviously they all have different limitations and vulnerable to many constructed attacks. For example, applying a sequence of image transformation will destroy many image watermarking schemes, as in StirMark [4].

Media watermarking is done usually by embedding watermarks in redundant bits such that it won't be perceived by human eyes. Software watermarking is following a similar concept that is a watermark can be embedded in sections of redundant code [4].
There are other protection techniques besides software watermarking. Those techniques are such as using registration database, following patent law, cryptography with hardware support, Obfuscation and Tamperproofing [14].

3.1 Hardware Techniques

Commonly, hardware-based protection approaches are based on using tokens. In this case, the execution of software programs is made dependent on a specific hardware element. Examples of hardware components are: CDs, dongles, smart cards, and so on. The dependability level between the hardware and software varies and can be strong or weak. If the software simply checks for the existence of the hardware token, then this

type of link between software and hardware is called weak. On the other hand, if the software cannot run without being linked to the hardware then the link is strong [1].

Hardware tokens are using physical dongle that should exist while running the software. Hardware tokens also can use distinctive characteristics of the floppy disk or CD to be the token that checks the program during its execution. For example, the toke can be based on the timing variances of the medium. This scheme will somewhat prevent copying because of the existence of distinctive characteristics [18].

3.2 Software Techniques

Software-based protection techniques are dependant on the same distributed software. Having the software itself as a protection model has many advantages such as increasing the distribution flexibility and reducing the protection added cost. Commonly used approaches for software protection, software-based approach, are by using a license key, a license file, and online or distributed activation code [1].

Even future software protection schemes will be vulnerable to attacks because they must be relying on finite-state machine to run the program, which can be examined and modified. For example, Windows XP, which has online activation and CD, was cracked and it took few months only to create a key generator for its activation. This means that software protection concern should not be on whether it’s going to be broken or not.

Instead, it should consider the time it will take to be broken and the possible consequences after being broken [1].

Insuring legal use of software can be done by keeping records of certified customers, using licensing information, linking the software to the hardware such as having a hardware movable dongle. One way to protect software against piracy is to use encryption techniques [7].

3.2.1 Encryption

One approach for protecting software is to use encryption. The idea here is to have the distributed software encrypted and a decryption key is needed to execute the software. Many encryption techniques can be used, such as having multiple encryption keys. A drawback of using encryption is the overhead it might add to the performance of the software [18].

3.2.2 Software Token

Software tokens are using the license file that applies checks while running the software. The token can also be obtained through connecting to the network [18]. A major drawback of having software tokens is that once the token is discovered, it is easy to search the internet for getting the serial number or license file information. For the online based token, its main disadvantage is that it forces the legal user to connect his computer to the internet.

3.2.3 Software Aging

Software aging is another approach for protecting against software piracy. It depends in creating program updates. Two advantages of this technique are: the reduced usability of pirated copies because of not having updated versions, and the need to frequently contact the pirate which increase the possibility of being caught [1].

3.2.4 Obfuscation and Tamper-proofing

Tamper-proofing and obfuscation are not software protection techniques by themselves. However, they are used in cooperation to increase the protection of other mechanisms. Tamper-proofing makes it difficult to remove the embedded protection message by making it hard to modify the program. Obfuscation hides the location of the embedded protection message by making it difficult to analyze the program [1].

3.2.5 Watermarking

A very common example to illustrate software watermarking is the following. Alice built software and sold it to Bob. Bob tries to make a pirate copy while Alice tries to copyright the software, by at least being able to prove ownership of a given copy, which may help in reducing piracy attacks [15].

Digital watermarks can be either visible or invisible. Visible watermarks prove the ownership by directly displaying the watermark transparently to everybody, as in displaying the logo of a TV channel on the screen corner. Invisible watermarks preserve hidden ownership information such as the source, author, creator, owner, distributor, and so on. Extracting the invisible watermark needs special detection software [7].

In watermarking, the pirated copy can be traced back to find the source of the illegal copy by looking into the watermark [18].

Steganography stands for hiding a secret message in a cover (ordinary) message, in order to be extracted at a destination. It basically allows for secret communication. Cryptograph, on the other hand, aims to hide the message contents, instead of trying to hide the message existence. Examples of Steganography are such as using invisible ink, hidden tattoos, microdots, and others [4].

Steganography have been used in media watermarking as in embedding invisible copyright information in a host images, audio, video, or even text [4]. “Steganography is the art and science of communicating in a way which hides the existence of the communication.” Watermarking is a special case of Steganography [8].

Software watermarking and fingerprinting are techniques used for protecting intellectual property and not for preventing copying the software. Basically they discourage the pirate by raising the probability of tracing the pirated copies. The basic idea is to embed an identification message in the original copies to identify the owner of the software [1].

In fingerprinting, the attacker can not insure that the software is totally cracked and the fingerprint is completely removed and no other fingerprints exist in the code. Yet a straightforward disadvantage of fingerprinting is that it assumes the ideal legal measures [1].

Software watermarking is different than software encryption. In encryption, a public-key encryption is constructed which require the decryption key in order to make the software files usable. After decrypting the software, it will become open without any encryption effect [7].

In software watermarking, the software can be open or usable, contrasting what happens in the encryption. The main objective of watermarking is not to prevent executing or using illegal pirated copies, however, its target is to prove and authenticate the ownership of the software by hiding a copyright message within the code of the software itself [7].

CHAPTER 4

STUDYING WATERMARKING WITH JAVA

4.1 Why Java?

Watermarking **Java** classes has the problem of being easy to decompile by the adversary. On the other hand, Java has the advantage of the reliable integrity of heap-allocated data structures [2] [3].

Studying any software watermarking scheme requires that we understand the language structure in which the watermark will be embedded, the way of embedding and extracting the watermark, possible kinds of attacks, and the overall cost of adding the watermark [4]. Any watermarking technique has three basic concerns: watermark size and its fraction of the program size, the form of the distributed program, and the expected different types of possible attacks [4].

We assume that the distributed object is in the form of a jar file containing set of Java class files. One trivial problem is that Java class files are easy to decompile and analyze.

However, Java classes have many factors that make it easier than others, like the integrity of heap allocation and the non editable executing code [4].

Java classes are running in virtual machine independent of hardware, which makes Java easily decompiled and reverse engineered. In order to hide the original Java classes and make it difficult to be obtained, several techniques can be applied such as code obfuscation, transformation, and watermarking [7].

Java uses bytecode and has the advantage of being portable. A disadvantage of bytecode is that it allows decmopilation to get the source code, which increases the probability of breaking the copyright and pirating the program. Java has the problem of piracy and hence, it became important to protect copyrights of Java programs [8]. Watermarking is basically a method of proving copyrights since the Java source code can never be prevented form being copied [8].

4.2 Java Virtual Machine and Byte code

Java class file has many sections: constant pool table, method table, and line number table [4]. More details about the Java Virtual Machine structure and functionality can be found in the website of sun Microsystems, Inc. http://www.sun.com/java/.

4.3 Sandmark

A very useful tool in the area of software protection algorithms (code obfuscation, software watermarking, and tamperproofing) is the SandMark, with more than 120,000

lines of Java code [http://sandmark.cs.arizona.edu]. It has many implemented algorithms, reverse engineering tools, and software complexity metrics [4].

Normally, a software watermarking algorithm takes, as an input, a jar-file and produces, as an output, a new jar-file. The jar-files contain class files (Java bytecode). SandMark architecture contains a number of plug-ins such as BCEL, DynamicJava, and BLOAT for bytecode editing, scripting, and code optimization in sequence [4].

The example below shows the effect of applying multiple obfuscating transformations done by using the SandMark tool [4].

```
public class C {
        static int gcd ( int x , int y) {
           int t ;
           while ( true ){
                boolean b = x % y == 0 ;
                if ( b ) return y ;
                t = x % y ; x = y ; y = t ;
           }
        }
        public static void main ( String [ ] a ){
                System.out.print("Answer : " ) ;
                System.out.println ( gcd ( 100 , 10 )) ;
        }
}
```

Figure 5: Example code before applying obfuscation techniques.

The applied transformations are:

	Obfuscation type	Effect
1	Boolean splitting	Splitting a Boolean variable into two small integer variables.
2	basic block splitting	Protecting bogus branch by inserting an opaquely false predicate.
3	string encoding	"Answer:" is encoded into a meaningless string to be decoded at run time.
4	scalar promotion	Converting integers to *java.lang.Integer*
5	signature unification	Giving all possible methods the same *Object[]* signature.
6	name obfuscation	"get0" replaced "gcd"

Table 1: List of applied obfuscation techniques.

Please refer to appendix A for more information about SandMark.

```
public class C {
 static Object get0(Object[] I) {
  Integer I7, I6, I4, I3; int t9, t8;
  I7=new Integer(9);
  for (;;) {
   if (((Integer)I[0]).intValue()%((Integer)I[1]).intValue()==0)
       {t9=1; t8=0;} else {t9=0; t8=0;}
   I4=new Integer(t8);
   I6=new Integer(t9);
   if ((I4.intValue()^I6.intValue())!=0)
     return new Integer(((Integer)I[1]).intValue());
   else {
     if ((((I7.intValue()+ I7.intValue()*I7.intValue())%2!=0)?0:1)!=1)
        return new Integer(0);
     I3=new Integer(((Integer)I[0]).intValue()%((Integer)I[1]).intValue());
     I[0]=new Integer(((Integer)I[1]).intValue());
     I[1]=new Integer(I3.intValue());
    }
  }
 }
 public static void main(String[] Z1) {
    System.out.print((String)Obfuscator.get0(
        new Object[] {(String)new Object[] {
"\u00AB\u00CD\u00AB\u00CD\uFF84\u2A16\u5D68\u2AA0\u388E\u91CF\u5326\u5604"
        }[0]}));
    System.out.println(((Integer)get0(new Object[]
         {(Integer)new Object[] {new Integer(100),new Integer(10)}[0],
         (Integer)new Object[] {
             new Integer(100),new Integer(10) }[1]})).intValue());
    }
}
```

Figure 6: Code after the effect of applying multiple obfuscation transformations.

CHAPTER 5

COMPARATIVE STUDY ON DIFFERENT WATERMARKING TECHNIQUES

5.1 Definition

A formal Model of Watermarking [2: 4]. Definition of software watermarking:

"Software *watermarking* that consists in *embedding* (that is the indelible unobtrusive fixing of invisible *stegosignatures*1 or *watermarks*, such as cryptographic signature and timestamp, in subject programs) and *extraction* (that is the detecting) of the stegosignatures) embedded in the *stegoprograms* (that is watermarked program sources). ("stego-xxx" means "xxx" in the context of hiding some embedded secret information.)" [7].

5.2 watermarking Classifications

According to [4], we can classify different software watermarking functions as follows:

1- *embed*(*P*;*w*;*key*) → *Pw*

By using a secret *key* and embedding a watermark *w*, we can transform a program *P* into *Pw*.

2- *extract*(*Pw*;*key*) → *w*

With the function *extract*, we can get the watermark *w* from *Pw*.

3- *recognize*(*Pw*;*key*;*w*) → [0:0;1:0]

If we use the *recognize* function, the returned value could reflect the probability of the existence of the watermark *w* in *P*.

4- *attack*(*Pw*) → *P'w*

The purpose of *attack* function is to alter the program *Pw* such that *w* can no longer be extracted.

The technique of embedding different watermark in every message is called **fingerprinting**. By using fingerprinting, we can trace the chain of the attacked copy and find the adversary. However, a trivial problem with fingerprinting is the vulnerability to collusion attacks, where the adversary compares different fingerprinted copies to find the location of the fingerprints [2] [3].

One type of watermarking, called fingerprinting, is simply embedding different secret message in every distributed object. The main advantage of fingerprinting is that it helps in tracing the source of theft in addition to detecting the theft incidence. Usually a fingerprint includes an identification number referring to product, seller, and the buyer. A very basic attack to fingerprinting is the collusion attack. This is done by simply

obtaining several copies of the program and applying set of comparisons until the fingerprints are located [4].

Watermarking classifications in [14] is as follows:

Type	**Description**	**Visibility**	**Robustness**
Authorship Mark (AM)	It embeds information identifying the author in the software	visible	robust
Fingerprinting Mark (FM)	It embeds information identifying the serial number or purchaser of the software	invisible	robust
Validation Mark (VM)	It embeds information verifying that the software is not changed from the originally authored	visible	fragile
Licensing Mark (LM)	It embeds information to control the way of using software	invisible	fragile

Table 2: Classification of watermarking techniques according to [14].

Watermarking classifications [20]:

Type	**Description**	**Visibility**	**Robustness**
Assertion Marks	Used to publicly claim the ownership of certain software	visible	robust
Prevention Marks	Focuses in preventing unauthorized users	invisible	robust
Affirmation Mark	Works as a seal of authoentcity	visible	fragile
Permission Mark	Should become invalid or illegible whenever there is a change or copy	invisible	fragile

Table 3: Classification of watermarking techniques according to [20].

5.3 Software Watermarking Actors

When we talk about software watermarking, we should consider four different parties: software producers, distributors, consumers, and adversaries [14].

Parties involved in software distribution are:

- Software provider: targeting maximizing profits.
- Legal user: paying for the software without searching for illegal copies.
- Pirate: have technical skills, target to break protection mechanisms with minimum risk of being caught.
- Illegitimate users: not technically skilled and don't want to pay for original copy [1].

5.4 Static and Dynamic Watermark

In general, there are two main approaches that can be used in software watermarking. The first approach is to make the watermark as part of the program behavior itself. For example, Easter Egg watermarking, where the watermark is displayed once a specific input is entered. The other approach is to embed the watermark in the program data structure, which can be traced while executing the program. An example of the second approach is the idea, by Collberg and Thomborsen, to embed a number as a graph that can be built as object structure while executing the program. Because of building the object structures dynamically, it is difficult to analyze during program execution, which involves flow analysis and pointer analysis. Therefore, it's too difficult for the attacker to locate the watermark statically [15].

An easy way to construct, embed, and extract a watermark is by adding static data watermarks. Dynamic watermarks are different than static watermarks since extracting the watermark is done while running the program and not by searching the data/code of the program. The idea here is that when the application is running with a certain input sequence, as a secret key owned by the software author, then the watermark can be extracted from the program's execution state [4].

5.4.1 Static Watermark

Static watermarks are stored in any section of the Java class file. Two static watermarking types are basically: **code watermarks**, in the executable instructions, and **data watermarks**, such as headers, and string sections [2] [3].

Data watermarks are easy to embed and extract, and hence, considered to be common. However, it is very vulnerable to distortive attacks such as obfuscation. For example, by splitting all strings into scattered substrings or by converting static data into a program distortion will happen [2] [3].

Code watermarks contain redundant information, similar to media watermarks having embedding in redundant bits. There are many simple distortive de-watermarking attacks and code obfuscation techniques that attack code watermarks. For example, flow-of-control can be destroyed by inserting predicated branches to break the basic block order [2] [3]. The watermark is

stored in the program itself as data or code. It can then be extracted directly from the program without executing it [7].

5.4.2 Dynamic Watermark

Static watermarks can be easily attacked by semantics-preserving transformations. Less work was done in the area of dynamic watermarking. In dynamic watermarks, the watermark is stored in the execution (behavior) of the program and not in the program itself. Therefore, dynamic watermarks have fewer threats of obfuscation transformations [2] [3].

Basically, there are three kinds of dynamic watermarks: Data Structure Watermark, Execution Trace Watermark, and Easter Egg Watermark. Those three methods differ in the way of storing and extracting the watermark. However, in all of the three methods, the application runs with a predetermined input sequence to enter the watermark state [2] [3].

Three common dynamic watermarks are: Dynamic Easter Egg Watermarks, Dynamic Execution Trace Watermarks, and Dynamic Data Structure Watermarks. Dynamic Easter Egg Watermarks is trivial and easily noticeable by the user since it basically displays the watermarking message or image after entering a certain input. If the watermark is extracted by tracing the addresses or instructions while executing the program, with a special input, then it is called Dynamic Execution

Trace Watermarks. Finally, Dynamic Data Structure Watermarks can be extracted by checking the values of particular program's variables with a particular input sequence, simply by using a debugger tool [4].

The watermark is stored in the execution state of the program. Three main types of dynamic watermarking techniques are: Easter egg watermarking, Dynamic data structure watermarking, and Dynamic execution trace watermarking [7].

5.5 Static Watermarking Techniques

Davidson and Myhrvold is the first published static software watermarking algorithm. It is an order-based algorithm where embedding the watermark is done by rearranging the order of the basic executing blocks. There are many possible attacks. A simple one is to randomly reorder the program basic blocks [4] [5].

Qu and Potkonjak software watermarking algorithm is based on renaming. The watermark is embedded in the program register allocation. It is based on renaming structures of the program. The major weaknesses are that it is easily attacked by decompilation/recompilation step and it has a low bit-rate [4] [5].

Static watermarks are embedded (or hidden) in the code or data of the program. The watermark is hidden in the redundant areas of the program, just like the multimedia watermarks, to be unpredictable.

Code watermarks can be stored in two different types of information:

1. Areas that don't have data dependencies or control dependencies. Such as in reordering case statements inside a switch statement or reordering control flow.
2. Alternate instructions, which have the same behavior. Java bytecode has many instructions that are equivalent.

Data watermarks are embedded in strings in areas that do not contain instructions, like in the constant pool.

Attacks such as optimization and obfuscation are major threats to static watermarks [8]. Moskowitz embeds a data watermark in an image. The image stored in the program static data section. A media watermarking algorithm is used to embed and extract the watermark. Distortion image attacks can be applied to this watermarking algorithm [4]. Arboit algorithm is based on embedding the watermark by adding a special opaque predicates to the program. Pattern matching is used for the watermark extraction. Trivial attacks by Pattern matching [4] [5].

Another static software watermarking algorithm was created by Stern. It uses a spread-spectrum technique for embedding watermark. The algorithm changes frequencies of certain instruction sequence by replacing them with equivalent sequence. The weaknesses of this algorithm include its vulnerability of being attacked by obfuscation to change the data-structures or data-encodings. Also it could be attacked by many low-level optimizations [4] [5].

In Monden algorithm, the watermark is encoded and embedded in a bogus method, guarded by a predicate always false, added to the program. This algorithm is vulnerable to optimization attacks [4].

5.6 Dynamic Watermarking Techniques

Problems of dynamic watermarking arise from the fact that it needs special input in order to extract the watermark. By using special tools that monitor program executions with some debugging techniques, the watermarks can be located and removed, or even destroyed. Program transformation techniques such as variable splitting or merging and program optimization techniques can also destroy dynamic data structure watermarking. These possible problems could threaten the research on this area (dynamic watermarking scheme) such that they are classified ineffective [7]. Dynamic watermarks are embedded in the program and can be generated while executing the program with certain input sequence [8].

5.6.1 Dynamic Graph Watermarking

Dynamic graph watermarking is one of the newly developed software watermarking technologies invented by Collberg and Thomborson in 1999 [2] [3].

One instance of the newly developed software watermarking technologies is dynamic graph watermarking (DGW) [Collberg and Thomborson 1999]. This

technology uses a dynamically created graph structure to represent a watermark message at the software execution time, instead of directly embedding a watermark message into the software program. The watermark number can be represented by the index of the watermark graph G in some convenient enumeration. The basic idea in the embedding function is that if number n is given; generate the n^{th} graph in the enumeration. The recognition function works similarly in such a way that if graph G is given, it extracts its index n in the enumeration. Both operations have to be efficient and hence, this technique cannot be used on generalized graphs since sub graph isomorphism is hard.

Collberg and Thomborson CT build the first dynamic watermarking algorithm. The watermark is embedded in the topology of a dynamically built graph structure. It is recognized at run time with a special input key. The main advantage of the CT algorithm is that it can overcome many obfuscation and optimization transformations [5].

Graph Theoretic Watermarking (GTW) has a high degree of stealth. It is resilient to edge-flip attacks or reordering basic blocks by having error-correcting graph techniques. However, the GTW has some weaknesses such as its dependability on the stability in recognizing the marked basic blocks while extracting the watermark. In addition, GTW is weak in resisting many semantics-preserving transformation attacks [5] The CT Algorithm [4:p.5].

Palsberg et al. is a dynamic watermarking algorithm based on CT. The watermark does not depend on a particular input sequence. The value of the watermark is represented by a planted planar cubic tree (PPCT) graph. Attacks to this algorithm can be through obfuscation transformations [4].

The watermark is embedded in the program by creating a graph structure that holds the watermark number. The dynamic graph watermarks method is stronger than other dynamic techniques because it has better resistance to transformation and other distortive attacks. Attacks to graph watermarks involve analyzing the program state, which is very difficult [8].

5.6.2 Data Structures

In dynamic data structure, the watermark is embedded in the state (global, heap, and stack data) of a program running with a certain input. Extracting the watermark is done either by examining current values of the application or by using debugger while running the program. An advantage of data structure watermarks is that the output does not immediately appear to an adversary. Also, it's difficult to locate the watermarks since only little information appears in the executable itself. The problem of data structure watermarks is that it is not immune to obfuscation attacks such as splitting or merging variables [2] [3].

Data structure watermarks differ since there is no specific output will be produced as a result of entering the key input into the program [4]. The watermark can be

obtained by examining definite signed program data that holds the watermark after entering a particular input values [7].

The watermark is embedded in the execution trace and it is not as easy to find as in the Easter egg watermarks. A watermark detecting tool is used to trace the program execution trace with a certain input sequence. The watermarks are independent to the program execution, and hence, are not difficult to locate. Those techniques are vulnerable to distortion attacks, such as obfuscation and variables or methods splitting. For example, a data structure watermark can be stored in the variables as shown below: [8]

```
char watermarks[];

watermarks[0] = 'c' ;
watermarks[1] = 'o' ;
watermarks[2] = 'p' ;
watermarks[3] = 'y' ;
watermarks[4] = 'r' ;
watermarks[5] = 'i' ;
watermarks[6] = 'g' ;
watermarks[7] = 'h' ;
watermarks[8] = 't' ;
watermarks[9] = '.' ;
```

Figure 7: Example of a data structure watermark.

5.6.3 Execution Trace

The watermark is embedded in the execution trace when the program is run with a certain input. Again, code obfuscation will also affect the execution trace [2] [3]. The watermark is stored in the execution trace of the program and only obtained for certain input sequence [7].

5.6.4 Easter Egg

If a code is watermarked in an Easter Egg, it performs a certain action such as displaying a copyright message or image only after the user enters a very unusual input. Unfortunately, the Easter Egg watermarks are easy to locate. Also, an adversary might generate a sequence of random input and wait for some strange output to be displayed [2] [3]. The watermark is extracted and displayed only after a certain input sequence is entered [7].

Easter Egg watermarks are easy to find. If the right input was discovered and entered then the watermark location can be easily traced by executing the program and using a debugging tool [4].

The watermark is generated and displayed as an output after entering a predefined input sequence. The watermark can be easily removed by an attacker once the right watermark is discovered [8] [23].

5.7 Software Watermarking by Diversity

One approach, presented in [1], is based in the idea of preventing software piracy through diversity. The scheme suggests that every installed copy of a program is uniquely different from others to insure that if this copy was successfully attacks, the same attack can not be generalized to other distributed copies. In order to escape from the static protection nature, the scheme also added a proposed continues

software updates, which makes the protection being of dynamic nature. If there are no updates on a particular distributed copy, then that copy is pirated. The advantage is that the pirate needs to be in synch with those continuing updates. Also, this scheme provides better control over the distributed copies. An analogy of applying diversity scheme in software piracy is the nature genetic diversity that provides protection against viruses and diseases [1].

The proposed scheme has several drawbacks. Simply, it is vulnerable to cracks and serials piracy attacks. Also, a copy of legally installed software can be easily obtained and installed in another personal computer, for example. Another drawback is its resilience dependability on updates. The issue of having diverse instance and multiple tailored updates adds another overhead. Moreover, the process of identifying legal and illegal users is another cost [1].

5.8 Other Software Watermarking Algorithms

Signal detection watermarking approach has been used in multimedia watermarking. The software watermarking can be designed using a similar approach of applying signal detection scheme to programs. Spread-Spectrum watermarking is a common watermark signal technique, such as the scheme using mutable instructions by Stern et al. The basic idea is to extract a vector r carrying

certain properties from the program. For example, it can show the graph depth for a specific point with a certain input sequence during program execution. An attack to this scheme can add to the depth of the program without affecting the original program operation. Like other software watermarking schemes, this scheme can be tamper-proofed and obfuscated in order to increase its defense level to attacks [8].

5.9 Attacks on Software Watermark

In media watermarking, most of the schemes are vulnerable to distortion attacks [2]. Three basic types of attacks can happen to a watermark: Subtractive attack, distortive attack, and additive attack. In the subtractive attack, the adversary tries to locate and crop out the watermark W. Distortive attack occurs when the adversary distort the watermark with an acceptable degraded quality. In the additive attack, the adversary adds his own watermark W', which overrides the original watermark W, or become impossible to detect W [2].

Any technique used to make de-watermarking attacks ineffective is called tamper-proofing [2]. Simple examples of such program transformation attacks are shown in the table below:

Watermarking type	Example	Program transformation attack
A comment	/* My software, version 1.0 */	Remove all comments
A data string	String v = " My software, version 1.0" ;	Split strings into shorter substrings
Order of instructions	n-branches or switch-statement	Reorder (insignificant) instructions
Idle instruction	Initializing unused string	Remove dead-code

Table 4: Examples of simple transformation attacks.

Other than the semantics-preserving program transformation, there are three other basic types of attacks that can happen to a watermark: Subtractive attack, distortive attack, and additive attack. In the subtractive attack, the adversary tries to locate and crop out the watermark W. Distortive attack occurs when the adversary distort the watermark with an acceptable degraded quality. In the additive attack, the adversary adds his own watermark W', which overrides the original watermark W, or become impossible to detect W [2] [3] [15].

Analyzing the text code is not the only way to attack a watermark. Observing the program behavior such as analyzing the state of the heap and registers during execution could help the attacker in finding the watermark [15].

Subtractive attacks can be protected by Tamperproofing and obfuscation, while randomization is used to protect against collusion attacks [15].

Three basic types of attacks are:-

1- **Subtractive attack:** the attacker manages to detect and crop out the watermark without affecting the original program.
2- **Distortive attack:** the attacker applies some distortive transformation attacks in a way that the watermark is being distorted. This will, of course, reduce the quality of the original program, but should be up to an acceptable level.
3- **Additive attack:** the attacker tends to insert one or more additional watermarks such that it's not possible to extract the original watermark, or at least to determine the priority timestamp [4].

The most common threatening attack is the distortive attack by applying semantic-preserving transformation. The goal is to make the watermark resistance to attacks such as decompilation, obfuscation, compression, and optimization [4].

There are many different kinds of attacks to software watermarking schemes and all the existing methods are susceptible to several attacks. A possible attack, but not reasonable, is to rewrite the complete application after studying its behavior. There are available tools that can help in doing translation, optimization, and obfuscation attacks. For example, a distortive obfuscation attack can split strings into substrings, which could make the watermark extraction more difficult. In fact, there are many software obfuscation methods that are considered threatening distortion attacks to software watermarks. Those transformations are like the following:

- Splitting/Merging a construct like a method or a variable.
- Increase/Decrease a construct dimension like an array.
- Change the nesting level of a construct.
- Redirect referencing like changing a method level of indirection.
- Rename a variable, method, or class construct.
- Reorder/Swap data statements or control dependencies.
- Clone a construct like duplicating a method [4].

Applying sequence of obfuscation techniques into a program will slow down, increase the size, and add overhead to the transformed program. Therefore, the attacker needs to insure that the overhead effects of transformations done into the program are acceptable. Otherwise, if the cost of applying such de-watermarking transformations into the speed and size of the program is so high, then they are not considered as attacks [4].

Software watermarking techniques are subject to several attacks such as:

Subtractive attacks: by locating and eliminating the existence of the watermark; like in removing static dead code or code protected by opaque predicates.

Preventing subtractive attacks can be done by making the program execution dependent on the watermark existence such that without the watermark the program will not be usable.

Distortive attacks: by applying code transformations to make extracting the watermark more difficult; like running obfuscation and optimization techniques. However, those

techniques can be considered, sometimes, as supportive to hide the watermark. Distortive attacks can be prevented by considering only part of the code or program data to extract the watermark. Yet, this does not guarantee that Distortive attacks became ineffective.

Additive attacks: by adding a new watermark to trim the value of the original watermark. To prevent such attacks, the original uniquely watermarked program can be compared with the attacked one.

Collusive attacks: by comparing different versions of the program where each is having different fingerprint. This can be done by monitoring program execution or statically analyze two or more versions of the programs. This attack can be prevented by allowing embedding more than one watermark. Also, one common watermark can be embedded in all the versions. In addition, more that one transformation can be applied to different versions to complicate the comparison process [7].

A main threat to watermarking schemes is the meaning-preserving transformation attack. This threat does not really count in media watermarking schemes [18]. A basic attack to protections approaches that use tokens will try to locate and remove the code that checks for the token existence by using a debugging tool. Also, the code that checks for license violation can be searched and removed. The target of software vendors using such approaches is to increase the cost of doing reverse engineering to there product [18].

5.10 Watermark Protection

Randomization, Obfuscation, and Tamperproofing are three powerful protection mechanisms that can greatly improve watermarking systems [15]. Code partitioning protection approach can be found in details in [18].

One obvious way to protect the watermark against attacks is to tamperproof and makes de-watermarking attacks ineffective. Distortion attacks cause difficulties in extracting the watermarks, as in cropping and compressing images, which made most media watermarking schemes vulnerable [4].

CHAPTER 6

OVERVIEW OF THE PROPOSED WORK

6.1 The need for Evaluation Benchmark

Although the research on software watermarking is still in its early stages and the work done in this area is comparatively low with reference to multimedia watermarking, Software watermarking received more attention recently and it is expected to even get more interest than what it was for the factors mentioned in earlier chapters. In fact, more than ten different software watermarking techniques were published in the past few years, regardless of there similarities and accreditation levels. Only few of them where studied, implemented, and experimented.

However, in all the work done in this area, none of them have used a specific or a particular evaluation measures to criticize the proposed software watermarking technique. Therefore, it's not clear yet how to practically evaluate and compare two or more software watermarking techniques. Having such evaluation standards and measures will perhaps lead to significant improvements on future research on software watermarking. One main advantage of evaluating a software watermarking technique is to allow for practical comparison among the available software watermarking techniques. Moreover,

having such benchmarking procedures will speed up the work and expedite the research progress in software watermarking area.

6.2 Reminiscent "Unrelated" Work

As usually being compared with multimedia watermarking, the work done in evaluating or benchmarking software watermarking is limited. In fact, there is some work on benchmarking multimedia watermarking approaches [12]. For example, StirMark benchmark 4.0 [16] is a well known benchmarking reference in the area of image watermarking. Fabien Petitcolas, one of the pioneer researchers in the area of multimedia watermarking, realized the importance of evaluating multimedia watermarks and proposed the first benchmark for multimedia watermarking schemes in his published paper Watermarking schemes evaluation in 2000 [17]. Other multimedia watermarking Benchmark tools are Certimark, Checkmark, and Optimark [24].

Unfortunately, during the study of existing software watermarking techniques and algorithms, we found that issues related to evaluating the proposed watermarking scheme were ignored or slightly addressed by some of there authors. In few papers, authors conducted some qualitative and quantitative experiments on there propose work. But yet, sometimes they neglected important evaluation factors or didn't consider certain measurements.

According to Christian Collberg and Clark Thomborson [3], designing software watermarks considers three issues: the size of the watermark with reference to the program size, the form of the program code, and the expected de-watermarking attacks. Reference [4] says that the quality of a Watermarking scheme depends on its response to different attacks.

In the paper titled “Experience with Software Watermarking” published in 2000, [15] software watermark quality is reflected by the degree of its resistance to piracy attacks. Those attacks are such that finding the location of the watermark, distorting it, and the ability of removing the watermark from the software. Experimenting can compare different programs by looking at code size before and after watermark is embedded, adding watermark time, retrieving watermark time, code execution time before and after watermark is embedded, and heap space size after watermark is embedded [15].

Christian Collberg and Clark Thomborson said in [2], three (trade-off) metrics determine the strength of a watermarked system: data-rate, stealth, and resilience. Data-rate represents the quantity of data hidden in the message. Stealth is a measure of how difficult for a user to recognize an embedded data. Resilience refers to the protection degree of the embedded message from an adversary.

According to paper Dynamic Graph-Based Software Watermarking [4], data rate, stealth, and resilience are three factors used to measure a watermarking scheme.

Data rate: number of bits of w / KB of cover message.

Stealth: measures w invisibility to the attacker.

Resilience: express w immunity degree to attackers.

Studies show trade-offs, i.e. high data rate implies low stealth and low resilience.

In [14], Jasvir Nagra, Collberg, and Thomborson wrote that different watermarking techniques can be compared with reference to the following properties: visibility, robustness, efficiency, and fidelity.

- **Visibility** measures the watermark level of unambiguous. It should look to how easy it is to distinguish and detect a watermark.
- **Robustness** is measured by considering the class of transformation after applying the watermarking algorithm. It is robust if the software program can survive after distortions.
- **Efficiency** considers the computation cost of adding the watermark in terms of time and memory usage.
- **Fidelity** is very much related to visibility. It measures the degree of effect that is caused by embedding a watermark into the original program.

Curran, Hurley, and Cinneide identified three main characteristics for an effective watermark are (with trade-offs):

Robustness: measures the resistance level of the watermark to attacks. Those attacks are such as optimization, decmopilation, obfuscation, and so on. The watermark should be present and extractable after the attack.

Capacity: measures the size of information embedded in the watermarking message.

Perceptibility: measures the visibility of the watermark. If the watermarked program has low quality in any regard, then the watermark is considered perceptible [8].

In short, most of the work done in software watermarking was looking for innovative embedding and extracting schemes. However, little work was done in evaluating those algorithms. In fact, there is no particular framework created for evaluating software watermarking algorithms.

6.3 Study Approach

According to Bender, (W. Bender, D. Gruhl, N. Morimoto, and A. Lu. Techniques for data hiding. *IBM Systems Journal*, 35(3&4):313. 336, 1996.), about media watermarking: "... all of the proposed methods have limitations. The goal of achieving protection of large amounts of embedded data against intentional attempts at removal may be unobtainable…" [4].

After looking into different software watermarking techniques with different embedding and recognition schemes, we need to clearly identify the important evaluation parameters. In this study we suggest an approach for evaluating and measuring software watermarking techniques for different aspects. Basically, numbers of quality factors are considered such as: embedding cost, extraction cost, running cost, the watermark itself, and the watermarking algorithm. The details of the important evaluation parameters are identified in the next section.

While size and time are important factors to consider, most software watermarking approaches consider types of attacks that change the program form and not its behavior. Those kinds of attacks, that intend to remove or distort watermarks embedded in the program structure, are called semantics-preserving program transformation. Examples of such attacks include: code optimization, dead-code removal, obfuscation, decompilation, and so on.

6.4 Proposed Evaluation Benchmarking

6.4.1 Size Efficiency (data rate/capacity)

Size efficiency considers the size of the embedded watermark with reference to the code of the application to be watermarked itself. It is called sometimes data rate or capacity. Data rate is the ratio of the added watermark P_w to the original code size P (P_w/P).

Measuring the size can be obtained by counting the lines of code, by measuring number of bits, or by measuring the heap size. Usually, the size is measured in terms of the number of bits. Of course, the larger the size of the embedded watermarks, the lower the robustness.

Another way to measure the size efficiency is to check the heap size while executing the watermarked and the un-watermarked applications under the same environment and input sequence rather than measuring

the static code size of the application. The basic idea is to compare the code size before and after embedding the watermark.

6.4.2 Time Efficiency

Time is a very critical factor in evaluating the effect of embedding the watermark in the original application. It is very important to insure that the applied watermarking scheme does not seriously degrade the execution time efficiency.

In general, time factor can be measured for three different portions: Execution time, Embedding time, and Extraction time. Embedding and extraction time can be directly measured during applying the watermarking scheme and are not affecting the watermarked application quality since the cost is only done once.

Thus, the major concerning factor is the Execution time, sometimes called embedding overhead. It is measured by finding the ration of execution time for an application with watermark being added T_w to the original execution time T. (T_w/T). Basically it measures running time before and after embedding the watermark.

6.4.3 Resilience Against Transformation Attacks (Robustness)

The objective is to measure how good different watermarking embedding algorithms are able to resist various attacks. It measures the degree to which the watermark is unfragile and resilience against transformation attacks (semantic-preserving transformations) such as code obfuscation and code optimization, register re-allocation, local variable splitting or merging, array splitting, class inheritance modification, basic block splitting (add nodes to control-flow), method merging (change control-flow graph), class encryption, code duplication (add nodes to control-flow), primitive boxing (change instructions in many basic blocks in a method), and so on.

Measuring the level of watermarking technique resilience against transformation attacks is very critical. The test should return a discrete value per any singular attack that is the attack result was success or fail. We assume that no partial watermark can be obtained as this indicates that the watermark is destroyed. After applying a set of tests for different application with different attacks, then the resilience level can be easily calculated.

6.4.4 Stealth (Invisibility/Perceptibility)

The stealth measures the degree to which the watermark is hidden and resists to being detected. Synonyms to stealth are watermark

invisibility and watermark perceptibility. Mainly, manual attacks, with the aid of some debugging tools, can help in locating the watermark itself or the watermark code from which the watermark can be retrieved.

Usually the stealth is not directly measured. First, the watermark is to be found. Then, it is to be removed. In fact, with the help of some advanced debugging tools it is not difficult to locate the watermark if compared with the original un-watermarked code. More over, it is not easy in many watermarking schemes to remove the watermark without affecting the application itself. However, measuring stealth is not straightforward since it depends on other factors such as having the original code, knowing the watermarking algorithm, and using advanced debugging tools.

6.4.5 Nature of the Watermarking Algorithm

The nature of the watermarking algorithm is a factor that depends on the application being watermarked itself. For example, if the watermarking algorithm is of high complexity and the application is very big in the execution time and size, then the overhead cost will be high. Also, the ability to understand or modify the algorithm could be considered.

However, in many cases, this factor is not of importance to the software producers as it does not affect the end users and pirates as well.

6.4.6 Other Factors

- Information embedded in the watermark

  Whether it is only a number, or it contains information such as copyright year, privilege granted, secret information, and so on.

- Source code/Byte code

  It looks for the watermark if it is embedded in the source code or the byte code. In most cases, the watermarking scheme is applied to the byte code for two reasons. First, it is easier having it in the compiled byte code rather than the need to recompile every class again. Second, normally the software organizations buy the applications in a class format from their producers. Therefore, they attempt to embed the watermark on the byte code directly rather than the need to go back to the original programmer to embed the watermark.

- Classes' Watermarked (one or all)

  It checks if the watermark is embedded to only one class, two classes, or all classes in the program, if parts of the application are

watermarked or as a whole. Also, it considers whether the watermark adds new classes or split classes or not.

- Number of keys

  Every algorithm requires one or more keys in addition to the watermark, which will be used for extracting the watermark. The larger the number of keys means it is easier to destroy by the attacker but difficult to discover the watermark.

Size	Number of Bits
	Lines of code
	Heap size
Time	Execution time
	Embedding time
	Extraction time
Resilience (against attacks)	Optimization
	Obfuscation
	Register re-allocation
	Local variable splitting/merging
	Class inheritance modification
	Basic block splitting
	Array splitting
	Method merging
	Class encryption
	Code duplication
	…
Stealth	Find W

	Remove W
Nature Watermarking Algorithm	Understandable
	Modifiable
Information embedded in the watermark	Numbers
	Characters
	Numbers and Characters
	Copyright Unique Information
	Privileges
	Others
Source code/Byte code	Source Code
	Byte Code
Classes' Watermarked	One
	More than one
	Modified classes
	Added classes
Number of keys	No keys
	One key
	Two keys
	More…

Table 5: Summary of proposed evaluation factors

CHAPTER 7

EXPERIMENTS AND RESULTS

7.1 Experimenting Tools and Environment

In our experiment, many different tools were used for studying different software protection algorithms. Large collection of watermarking schemes and attacks for Java bytecode were used. The tools include different text files, folders, archives, editors, decompilers, and other debugging and analysis tools.

For all the experimentation done in this study, following are the specifications of the used machine:

- Windows XP Professional
- Pentium 4
- CPU 2.66 GHz
- 1.00 GB of RAM
- JDK 1.4

7.2 Evaluation of Watermarks and Applications

Throughout the experiments, ten different watermarking techniques that are implemented in SandMark where basically used. The experiment could be done to any number of watermarking schemes and not limited only for the ones used in our study. In fact, during the experimentations some difficulties were faced while trying to execute specific algorithms for different cases. Nevertheless, those unsuccessful executions where noted and included as part of the results. The algorithms that require manual modifications on the source code where neglected. As stated earlier, the assumption is that the algorithms are applied to applications in bytecode format. The table below shows the briefing summary of the techniques used.

W1: String Constant
Embed a watermark in a string in the constant pool
W2: Stern
This algorithm (by Stern et. al.) embeds a static watermark spread throughout the body of the code as the frequency of occurrence of identified groups of instructions. See Help for restrictions on input.
W3: Register Type
HatTrick is a way of encoding watermarks based on special local variables that encode a message based on the locals' types. Each type maps to a base-10 digit that encodes a numerical watermark.
W4: Qu/Potkonjak
AssignLV is a watermarking algorithm that embeds the watermark in the local variable assignment by adding constraints to the interference graphs.
W5: Monden
Implements the watermarking technique described in A Practical Method for Watermarking Java Programs by A. Monden, H. Iida, K. Matsumoto, K. Inoue, and K. Torii. The watermark is embedded by replacing instruction in a dummy method, which is added to the application.
W6: Graph Theoretic Watermarking
Venkatesan's Graph Theoretic Watermarking Algorithm embeds the watermark in control flow graph within the program.
W7: AddMethField

AddMethField is a static watermarker which embeds the watermark by splitting it in half the first part becomes the name of a new field and the second becomes part of the name of a new method.
W8: Add Switch
Embeds a watermark in the labels of a switch statement
W9: Add Initialization
Add Initialization is a StaticWatermarker which embeds a numeric watermark by breaking it into 2-digit numbers and adding them to the constant pool
W10: Add Expression
This algorithm embeds a static watermark as a bogus expression that is assigned to a new local variable. The watermark is recognized with the help of the local variable name

Table 6: Watermarks techniques used in the experimentations

The obfuscation techniques used in the experiment are as shown in the table below:

Obfuscation	Description
O1: Field Assignment	The AddBogusFields obfuscator adds a bogus field to each class in an application and throughout the class makes assignments to the field.
O2: Dynamic Inliner	DynamicInliner inlines non-static methods, determining which branch to use at runtime.
O3: Duplicate Register	Takes a local variable in a method and splits references to it with a new variable (which stays synchronized).
O4: Constant Pool	ConstantPool Reorderer randomly reassigns constant pool indices.
O5: Class Encrypter	Class Encrypter encrypts class files and causes them to be decrypted at runtime.
O6: Array Splitter	Splits an array into 2 arrays, while preserving program semantics.
O7: Block Maker	Use a BasicBlockMarker to mark basic blocks randomly. This is a useful against some watermarking algorithms
O8: Bludgeon	Converts all static methods to take Object[] and return Object.
O9: Bolean Splitter	This algorithm detects boolean variables and arrays and modifies all uses and definitions of these variables.
O10: Branch Inverter	This algorithm negates the if instruction in the ifelse statementand exchanges the if and else part of the body
O11: Class Splitter	ClassSplitter splits a class in half by moving some methods and fields to a superclass.

Table 7: Obfuscation techniques used in the experimentations

In the experimentations done, seven different applications were used with different purposes and varying number of classes and size. The applications were randomly selected to show reasonable results. All the applications are in an executable JAR format. For more details, see the table below.

Application	**No. of classes**	**Description**
XMLTree	27	A simple XML file editor. It displays XML nodes in a file in a tree on the left hand side of the program window, and displays the highlighted node's attributes on the right hand side.
TTT	4	The world's most famous game tic-tac-toe.
toy_1.4	124	Visual X-TOY simulator, a visually-appealing simulation of a PDP8-style machine and an IDE for writing programs in the TOY machine language
spiro	24	The Spiro applet is a tool for creating a certain kind of graphic, while keeping things reasonably simple and portable.
jdrill2_3_1	18	A quiz window tool program for learning program for testing and learning Japanese.
Cvt2Mae	28	A data format converter on various types of array data file formats.
Conzilla1.1Beta2	586	The second generation concept browser, a knowledge management tool with many purposes.

Table 8: Sample applications used in the experimentations

7.3 Experimentation and Testing

7.3.1 Size

No Watermark	
Application	**Code Size (KB)**
XMLTree	57.2
TTT	8.53
toy_1.4	573
Spiro	73.5
jdrill2_3_1	77.6
Cvt2Mae	326
Conzilla1.1Beta2	1557
Average :	381.83

Table 9: Size of applications without watermarks being embedded

Watermarked Code Size (KB)										
Application	**W1**	**W2**	**W3**	**W4**	**W5**	**W6**	**W7**	**W8**	**W9**	**W10**
XMLTree	62.2	62.5	63.6	62	62.8	91.4	62.3	62.3	62.2	62.2
TTT	9.2	9.48	9.48	9.1	10.4	x	9.32	9.3	9.23	9.21
toy_1.4	573	573	574	571	593	x	573	573	573	573
spiro	81.1	81.4	82.1	80.5	85	115	81.3	81.3	81.2	81.2
jdrill2_3_1	86	85.7	85.7	85.2	88.1	x	85.6	84.5	85.5	85.5
Cvt2Mae	326	326	327	324	328	x	326	326	326	326
Conzilla1	1566	1566	1566	1563	1568	x	1566	1566	1566	1566
Average :	386.21	386.30	386.84	384.97	390.76	103.20	386.22	386.06	386.16	386.16

Table 10: Size of applications after watermarks being embedded

Watermark Code Size (KB)										
Application	**W1**	**W2**	**W3**	**W4**	**W5**	**W6**	**W7**	**W8**	**W9**	**W10**
XMLTree	5	5.3	6.4	4.8	5.6	34.2	5.1	5.1	5	5
TTT	0.67	0.95	0.95	0.57	1.87	x	0.79	0.77	0.7	0.68
toy_1.4	0	0	1	-2	20	x	0	0	0	0
spiro	7.6	7.9	8.6	7	11.5	41.5	7.8	7.8	7.7	7.7
jdrill2_3_1	8.4	8.1	8.1	7.6	10.5	x	8	6.9	7.9	7.9
Cvt2Mae	0	0	1	-2	2	x	0	0	0	0
Conzilla1	9	9	9	6	11	x	9	9	9	9
Average :	4.38	4.46	5.01	3.14	8.92	37.85	4.38	4.22	4.33	4.33

Table 11: Size of watermarks being embedded

Increase Change (%)										
Application	**W1**	**W2**	**W3**	**W4**	**W5**	**W6**	**W7**	**W8**	**W9**	**W10**
XMLTree	11.76	12.47	15.06	11.29	13.18	37.42	12.00	12.00	11.76	11.76
TTT	1.58	2.24	2.24	1.34	4.40	x	1.86	1.81	1.65	1.60
toy_1.4	0.00	0.00	2.35	-4.71	47.06	x	0.00	0.00	0.00	0.00
spiro	17.88	18.59	20.24	16.47	27.06	97.65	18.35	18.35	18.12	18.12
jdrill2_3_1	19.76	19.06	19.06	17.88	24.71	x	18.82	16.24	18.59	18.59
Cvt2Mae	0.00	0.00	2.35	-4.71	4.71	x	0.00	0.00	0.00	0.00
Conzilla1	21.18	21.18	21.18	14.12	25.88	x	21.18	21.18	21.18	21.18
Average :	10.31	10.50	11.78	7.38	21.00	67.53	10.32	9.94	10.18	10.18

Table 12: Percentage of change after embedding watermarks

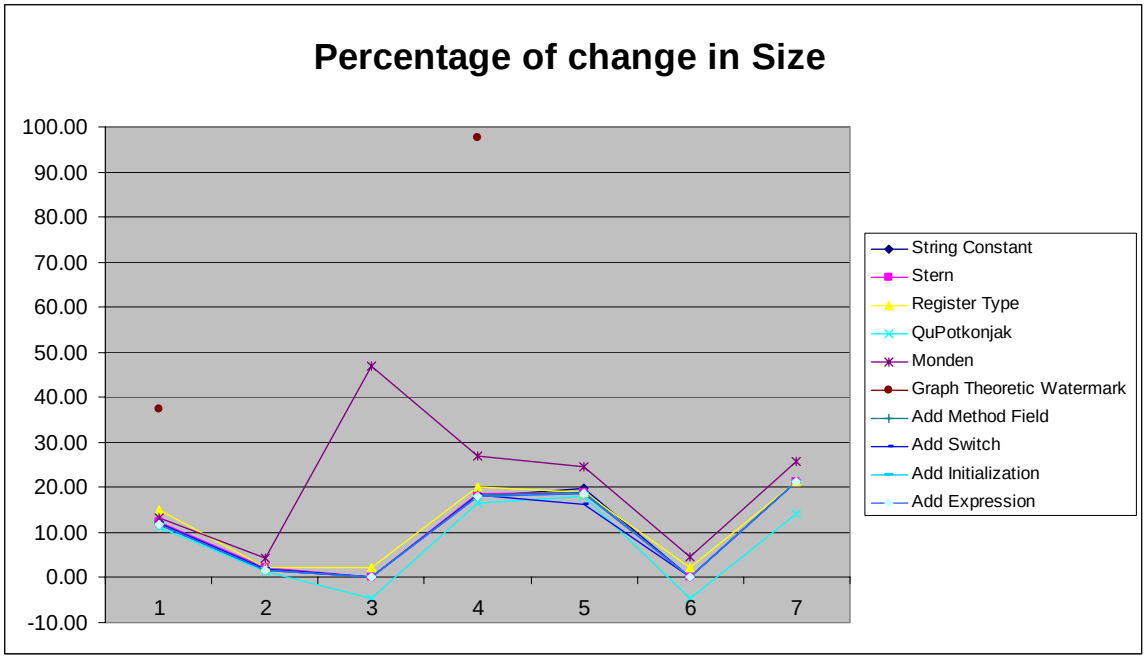

Figure 8: Percentage of change of size after embedding watermarks

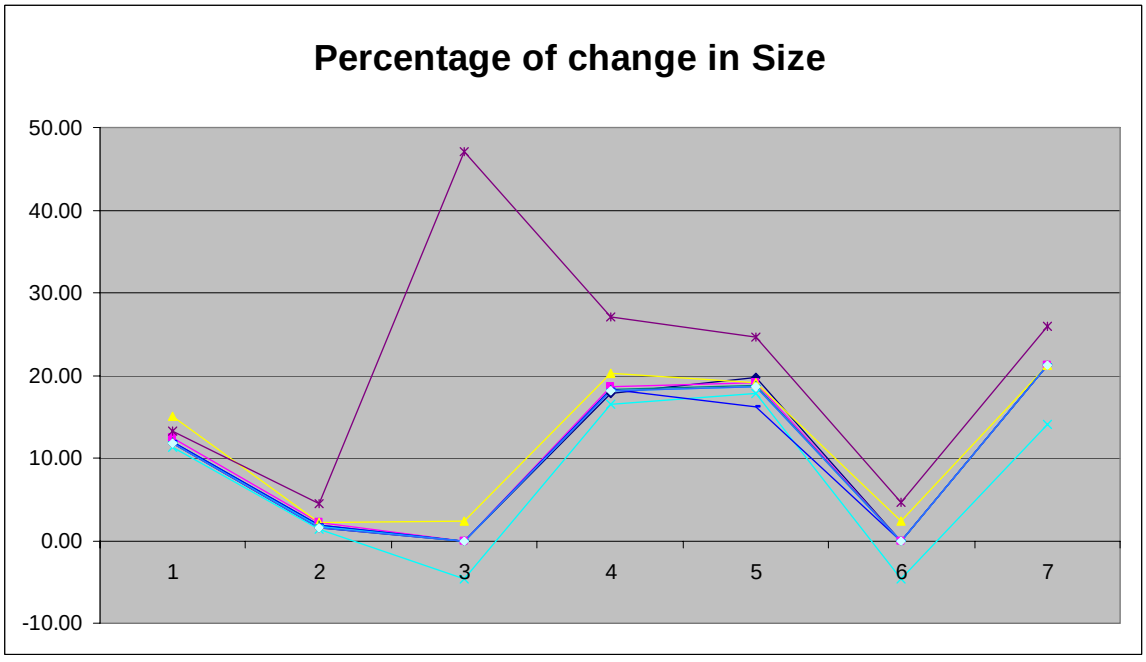

Figure 9: Percentage of change of size after embedding watermarks

7.3.2 Time

No Watermark	
Application	**Execution Time (ms)**
XMLTree	2.126
TTT	1.624
toy_1.4	5.456
spiro	0.801
jdrill2_3_1	1.43
Cvt2Mae	1.63
Conzilla1.1Beta2	25.12
Average :	**5.46**

Table 13: Execution time of applications without watermarks being embedded

Watermarked Execution Time (ms)										
Application	**W1**	**W2**	**W3**	**W4**	**W5**	**W6**	**W7**	**W8**	**W9**	**W10**
XMLTree	2.144	2.168	2.158	2.162	x	2.282	2.188	2.186	2.058	2.222
TTT	1.686	1.682	1.69	1.678	1.648	x	1.644	1.65	1.602	1.626
toy_1.4	5.57	5.618	5.43	5.8	6.12	x	5.628	5.508	5.542	5.822
spiro	0.868	0.906	0.871	0.861	0.945	0.924	0.904	0.925	0.896	0.889
jdrill2_3_1	1.416	1.374	1.352	1.432	1.438	x	1.392	1.356	1.402	1.4
Cvt2Mae	1.656	1.686	1.7	x	1.674	x	1.572	1.588	1.636	1.68
Conzilla1	24.98	25.14	24.78	25.23	25.14	x	24.86	25.61	25.12	25.38
Average :	**5.47**	**5.51**	**5.43**	**6.19**	**6.16**	**1.60**	**5.46**	**5.55**	**5.47**	**5.57**

Table 14: Execution time of applications after watermarks being embedded

Difference in Watermarked Execution Time (ms)										
Application	**W1**	**W2**	**W3**	**W4**	**W5**	**W6**	**W7**	**W8**	**W9**	**W10**
XMLTree	0.018	0.042	0.032	0.036	x	0.156	0.062	0.06	-0.068	0.096
TTT	0.062	0.058	0.066	0.054	0.024	x	0.02	0.026	-0.022	0.002
toy_1.4	0.114	0.162	-0.026	0.344	0.664	x	0.172	0.052	0.086	0.366
spiro	0.067	0.105	0.07	0.06	0.144	0.123	0.103	0.124	0.095	0.088
jdrill2_3_1	-0.014	-0.056	-0.078	0.002	0.008	x	-0.038	-0.074	-0.028	-0.03
Cvt2Mae	0.026	0.056	0.07	x	0.044	x	-0.058	-0.042	0.006	0.05
Conzilla1	-0.14	0.02	-0.34	0.11	0.02	x	-0.26	0.49	0	0.26
Average :	**0.02**	**0.06**	**-0.03**	**0.10**	**0.15**	**0.14**	**0.00**	**0.09**	**0.01**	**0.12**

Table 15: Execution time of watermarks being embedded

Increase Change (%)										
Application	**W1**	**W2**	**W3**	**W4**	**W5**	**W6**	**W7**	**W8**	**W9**	**W10**
XMLTree	0.84	1.94	1.48	1.67	x	6.84	2.83	2.74	-3.30	4.32
TTT	3.68	3.45	3.91	3.22	1.46	x	1.22	1.58	-1.37	0.12
toy_1.4	2.05	2.88	-0.48	5.93	10.85	x	3.06	0.94	1.55	6.29
spiro	7.72	11.59	8.04	6.97	15.24	13.31	11.39	13.41	10.60	9.90
jdrill2_3_1	-0.99	-4.08	-5.77	0.14	0.56	x	-2.73	-5.46	-2.00	-2.14
Cvt2Mae	1.57	3.32	4.12	x	2.63	x	-3.69	-2.64	0.37	2.98
Conzilla1	-0.56	0.08	-1.37	0.44	0.08	x	-1.05	1.91	0.00	1.02
Average :	**2.04**	**2.74**	**1.42**	**3.06**	**5.13**	**10.07**	**1.58**	**1.78**	**0.84**	**3.21**

Table 16: Percentage of change of execution time after embedding watermarks

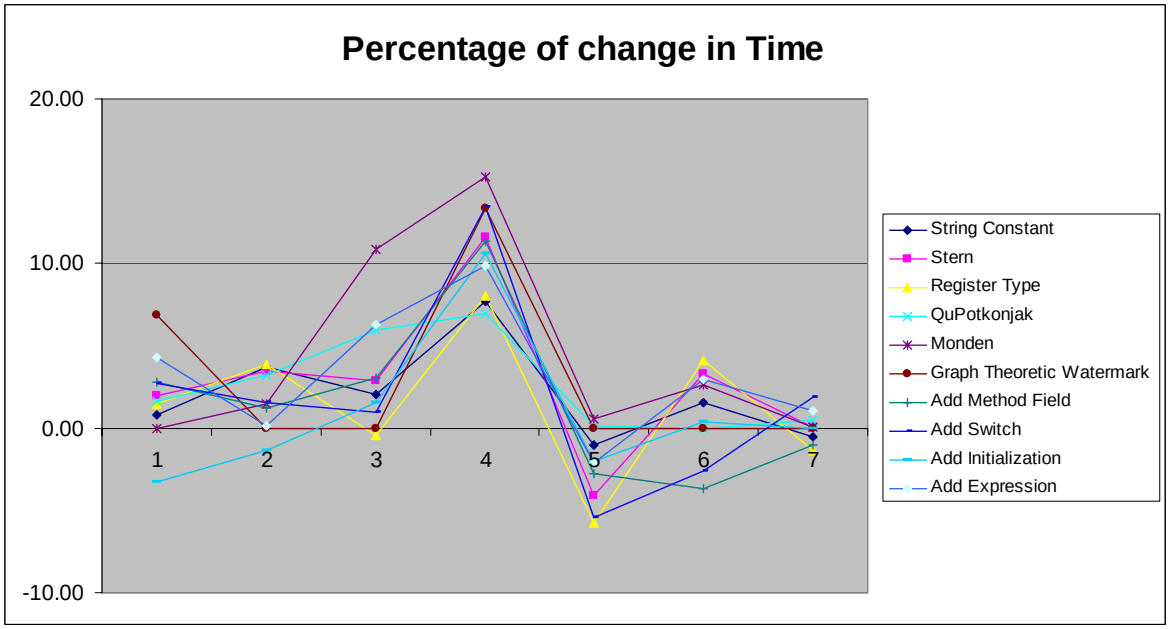

Figure 10: Percentage of change of execution time after embedding watermarks

7.3.3 Obfuscation Attacks

Results are exposed to be in binary measures as follows:

1: W found

0: W not found

x: Failed to execute

Obfuscation Attack: **Array Splitter (O6)**										
Application	**W1**	**W2**	**W3**	**W4**	**W5**	**W6**	**W7**	**W8**	**W9**	**W10**
XMLTree	1	1	1	0	1	0	1	1	1	1
TTT	1	1	1	0	1	x	1	1	1	1
toy_1.4	x	x	x	x	x	x	x	x	x	x
spiro	1	1	1	0	1	0	1	1	1	1
jdrill2_3_1	1	1	1	0	1	x	1	1	1	1
Cvt2Mae	x	x	x	x	x	x	x	x	x	x
Conzilla1	x	x	x	x	x	x	x	x	x	x

Table 17: Effect of Array Splitter Obfuscation attack

Obfuscation Attack: **Block Make (O7)**										
Application	**W1**	**W2**	**W3**	**W4**	**W5**	**W6**	**W7**	**W8**	**W9**	**W10**
XMLTree	1	1	1	0	1	0	1	1	1	1
TTT	1	1	1	0	1	0	1	1	1	1

Table 18: Effect of Block Make Obfuscation attack

Obfuscation Attack: **Bludgeon (O8)**										
Application	**W1**	**W2**	**W3**	**W4**	**W5**	**W6**	**W7**	**W8**	**W9**	**W10**
XMLTree	1	1	1	0	1	0	1	1	1	1
TTT	1	1	1	0	1	0	1	1	1	1

Table 19: Effect of Bludgeon Obfuscation attack

Obfuscation Attack: **Class Encrypter (O5)**										
Application	**W1**	**W2**	**W3**	**W4**	**W5**	**W6**	**W7**	**W8**	**W9**	**W10**
XMLTree	0	0	0	0	0	0	0	0	0	0
TTT	0	0	0	0	0	0	0	0	0	0

Table 20: Effect of Class Encrypter Obfuscation attack

7.3.4 Optimization Attacks

Optimization Attack: **Inliner: inlines static methods.**										
Application	**W1**	**W2**	**W3**	**W4**	**W5**	**W6**	**W7**	**W8**	**W9**	**W10**
XMLTree	1	0	0	0	1	x	1	1	1	0
TTT	1	0	0	0	1	x	1	1	1	0
toy_1.4	1	0	0	0	1	x	0	1	1	0
spiro	1	1	0	0	1	x	1	1	1	0
jdrill2_3_1	1	1	0	0	1	x	1	1	1	0
Cvt2Mae	1	0	0	0	1	x	0	1	1	0
Conzilla1	x	x	x	x	x	x	x	x	x	x

Table 21: Effect of Inliner optimization attack

Optimization Attack: **Variable Reassigner**										
Application	**W1**	**W2**	**W3**	**W4**	**W5**	**W6**	**W7**	**W8**	**W9**	**W10**
XMLTree	1	1	0	0	1	0	1	1	1	0
TTT	1	1	0	0	1	x	1	1	1	0
spiro	1	1	0	0	1	0	1	1	1	0

Table 22: Effect of Variable Reassigner optimization attack

7.3.5 Collusive Attacks

Ability to recognize first Watermark if the same algorithm is applied twice										
Application	**W1**	**W2**	**W3**	**W4**	**W5**	**W6**	**W7**	**W8**	**W9**	**W10**
XMLTree	1	1	0	0	1	x	1	0	0	1
TTT	0	1	1	0	1	x	0	0	0	0
toy_1.4	1	1	1	0	1	x	0	0	0	1
Spiro	1	0	0	0	1	x	1	0	0	1
jdrill2_3_1	0	1	1	0	1	x	0	0	0	0
Cvt2Mae	0	1	1	0	1	x	0	0	0	0
Conzilla1	0	0	1	0	0	x	0	0	0	0

Table 23: Effect of adding the same watermarking algorithm twice to the first watermark

Ability to recognize second Watermark if the same algorithm is applied twice										
Application	**W1**	**W2**	**W3**	**W4**	**W5**	**W6**	**W7**	**W8**	**W9**	**W10**
XMLTree	0	1	1	0	0	x	0	1	1	0
TTT	1	1	0	0	0	x	1	1	1	1
toy_1.4	0	1	0	0	0	x	0	1	1	0
spiro	0	0	1	0	0	x	0	1	1	0
jdrill2_3_1	1	1	0	0	0	x	1	1	1	1
Cvt2Mae	1	1	0	0	0	x	1	1	1	1
Conzilla1	1	0	0	0	0	x	1	1	1	1

Table 24: Effect of adding the same watermarking algorithm twice to the 2nd watermark

Ability to recognize the original Watermark if a different algorithm is applied										
Application	**W1**	**W2**	**W3**	**W4**	**W5**	**W6**	**W7**	**W8**	**W9**	**W10**
XMLTree	1	1	1	0	1	0	1	1	1	1
TTT	1	1	1	0	1	x	1	1	1	1
toy_1.4	1	1	1	0	1	x	0	1	1	1
spiro	1	1	1	0	1	0	1	1	1	1
jdrill2_3_1	1	1	1	0	1	x	1	1	1	1
Cvt2Mae	1	1	1	0	1	x	0	1	1	1
Conzilla1	1	0	1	0	0	x	0	1	1	1

Table 25: Effect of adding different watermarking algorithm

7.3.6 Manual Attacks

Results of experimenting “spiro” application only.										
	W1	**W2**	**W3**	**W4**	**W5**	**W6**	**W7**	**W8**	**W9**	**W10**
Searching for the W	0	1	0	2	1	1	1	1	1	1
Ability of removing W	3	1	3	3	2	3	1	1	1	1

Table 26: Effect of Manual attacks on “spiro” application

Where:

0: not found
1: easily
2: moderate
3: hardly

7.4 Analysis and Observations

7.4.1 Size

The percentage of change of size on the effect of watermarking is almost following the same trend with minor difference. The majority of watermarks has expansion rate of about 10% in size. The maximum tested result has increase in size of about 67%. Levels of size expansion can be simply classified according to the table below:

	Level 1	Level 2	Level 3	Level 4	Level 5
Expansion in Size	< 10 %	10 – 20 %	20 – 50 %	50 – 100 %	> 100 %

Table 27: Classification based on Expansion in Size

7.4.2 Time

As seen in the results above, the percentage of change of time on the effect of watermarking is almost following the same trend with minor differences for some applications and techniques. However, unlike the expansion in size, the rate of change of execution time was comparatively low and some results showed even negative values for certain applications. This is expected since applying the scheme might have an overhead in time during embedding or recognizing the watermark and not while executing the watermarked program. Nonetheless, the main concern is the efficiency of the watermarked applications.

	Level 1	Level 2	Level 3	Level 4	Level 5
Expansion in Time	< 1 %	1 – 5 %	5 – 10 %	10 – 20 %	> 20 %

Table 28: Classification based on Expansion in Time

By observing size and time results the following results can be concluded:

- Almost the same trend between time and size is conserved no matter what technique of watermarking is used.

No Watermarking

No	**Application**	**Size(KB)**	**Time(s)**	**Time(ms)**
1	spiro	73.5	0.801	80.1
2	jdrill2_3_1	77.6	1.43	143
3	TTT	8.53	1.624	162.4
4	Cvt2Mae	326	1.63	163
5	XMLTree	57.2	2.126	212.6
6	toy_1.4	573	5.456	545.6
7	Conzilla1.1Beta2	1557	25.12	2512
	Average	381.83	5.46	545.53

Table 29: Size over time ratio with no watermarks

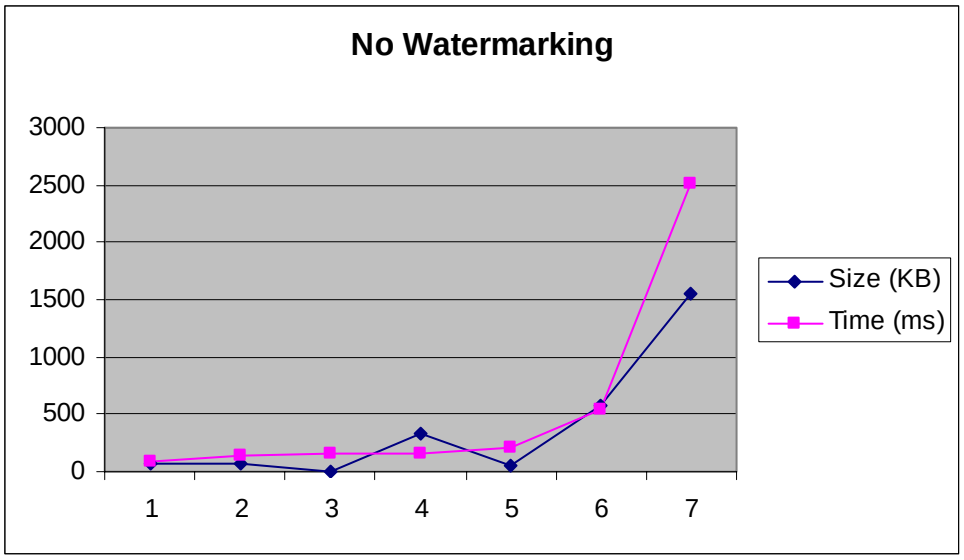

Figure 11: Size over time ratio with no watermarks

String Constant

No	**Application**	**Size(KB)**	**Time(s)**	**Time(ms)**
1	spiro	81.1	0.868	86.8
2	jdrill2_3_1	86	1.416	141.6
3	TTT	9.2	1.686	168.6
4	Cvt2Mae	326	1.656	165.6
5	XMLTree	62.2	2.144	214.4
6	toy_1.4	573	5.57	557
7	Conzilla1.1Beta2	1566	24.98	2498
	Average	386.2143	5.474286	547.4286

Table 30: Size over time ratio with String Constant watermark

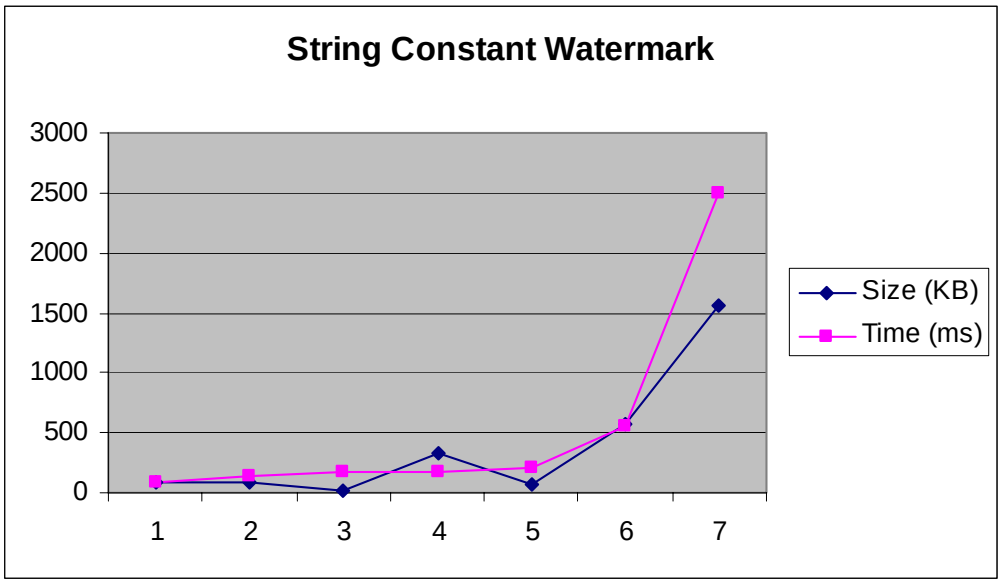

Figure 12: Size over time ratio with String Constant watermark

Stern Watermark

No	**Application**	**Size(KB)**	**Time(s)**	**Time(ms)**
1	spiro	81.4	0.906	90.6
2	jdrill2_3_1	85.7	1.374	137.4
3	TTT	9.48	1.682	168.2
4	Cvt2Mae	326	1.686	168.6
5	XMLTree	62.5	2.168	216.8
6	toy_1.4	573	5.618	561.8
7	Conzilla1.1Beta2	1566	25.14	2514
	Average	386.2971	5.510571	551.0571

Table 31: Size over time ratio with Stern watermark

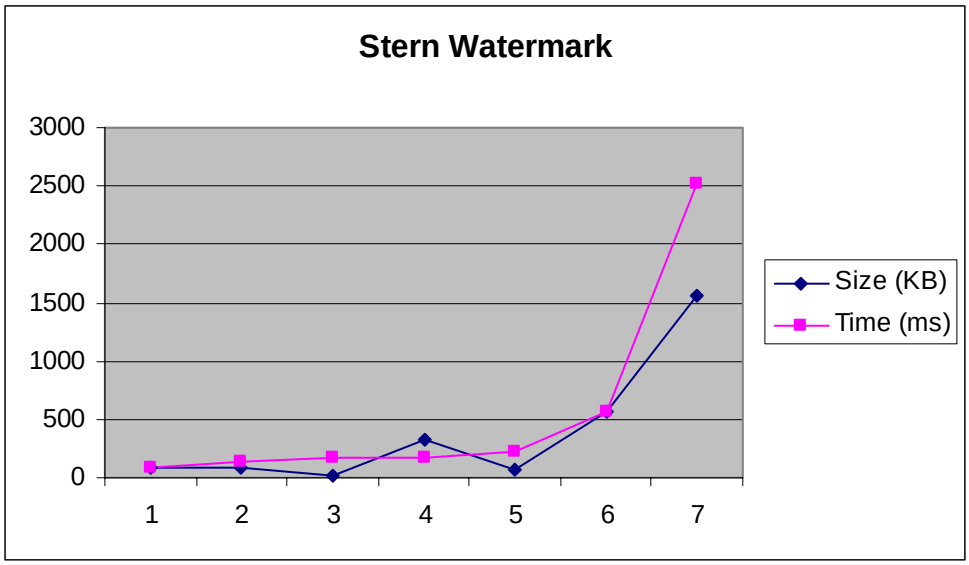

Figure 13: Size over time ratio with Stern watermark

Register Type

No	**Application**	**Size(KB)**	**Time(s)**	**Time(ms)**
1	spiro	82.1	0.871	87.1
2	jdrill2_3_1	85.7	1.352	135.2
3	TTT	9.48	1.69	169
4	Cvt2Mae	327	1.7	170
5	XMLTree	63.6	2.158	215.8
6	toy_1.4	574	5.43	543
7	Conzilla1.1Beta2	1566	24.78	2478
	Average	5.425857	386.84	542.5857

Table 32: Size over time ratio with Register Type watermark

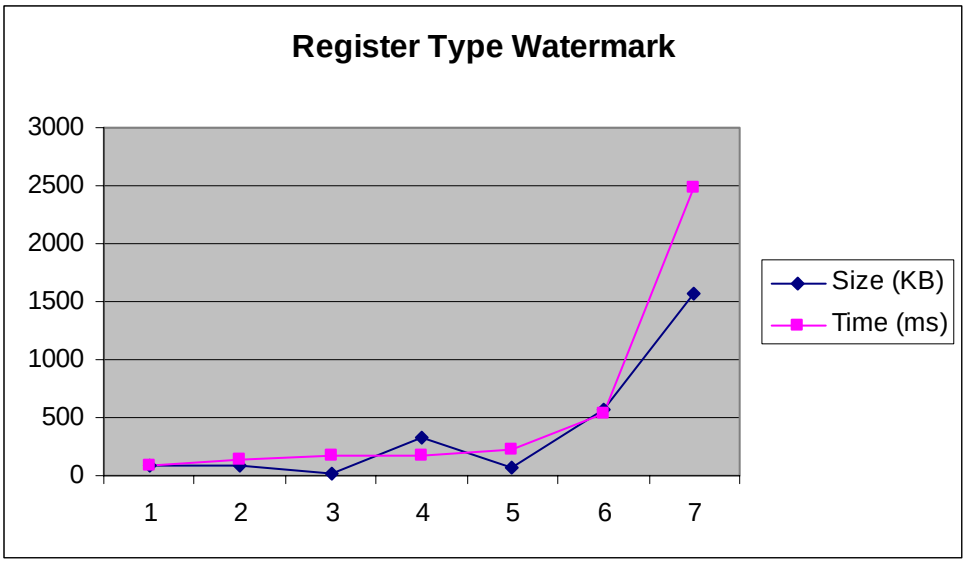

Figure 14: Size over time ratio with Register Type watermark

QuPotkonjak

No	**Application**	**Size(KB)**	**Time(s)**	**Time(ms)**
1	spiro	80.5	0.861	86.1
2	jdrill2_3_1	85.2	1.432	143.2
3	TTT	9.1	1.678	167.8
4	Cvt2Mae	324	x	x
5	XMLTree	62	2.162	216.2
6	toy_1.4	571	5.8	580
7	Conzilla1.1Beta2	1563	25.23	2523
	Average	6.193833	384.9714	619.3833

Table 33: Size over time ratio with QuOotkonjak watermark

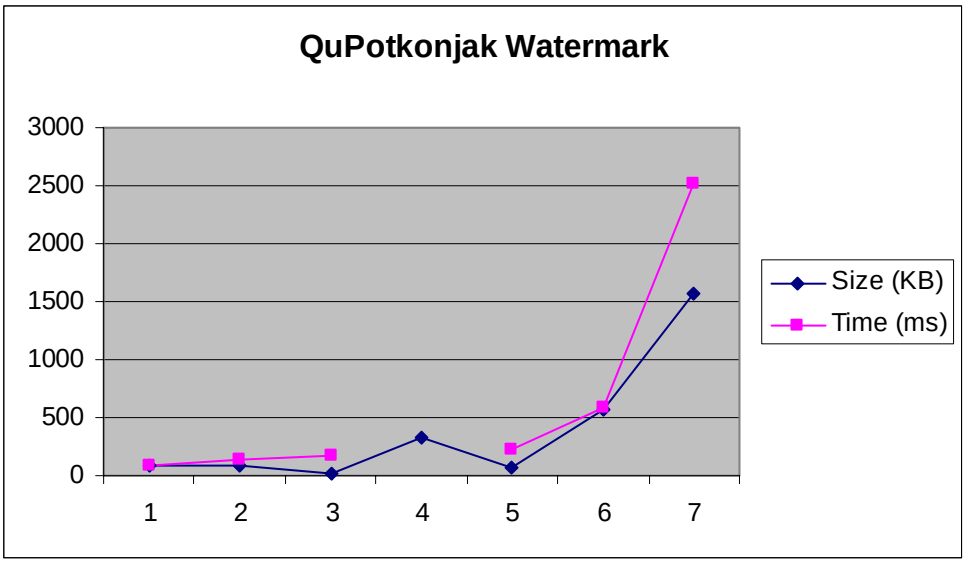

Figure 15: Size over time ratio with QuOotkonjak watermark

Monden

No	**Application**	**Size(KB)**	**Time(s)**	**Time(ms)**
1	spiro	85	0.945	94.5
2	jdrill2_3_1	88.1	1.438	143.8
3	TTT	10.4	1.648	164.8
4	Cvt2Mae	328	1.674	167.4
5	XMLTree	62.8		
6	toy_1.4	593	6.12	612
7	Conzilla1.1Beta2	1568	25.14	2514
	Average	6.160833	390.7571	616.0833

Table 34: Size over time ratio with Monden watermark

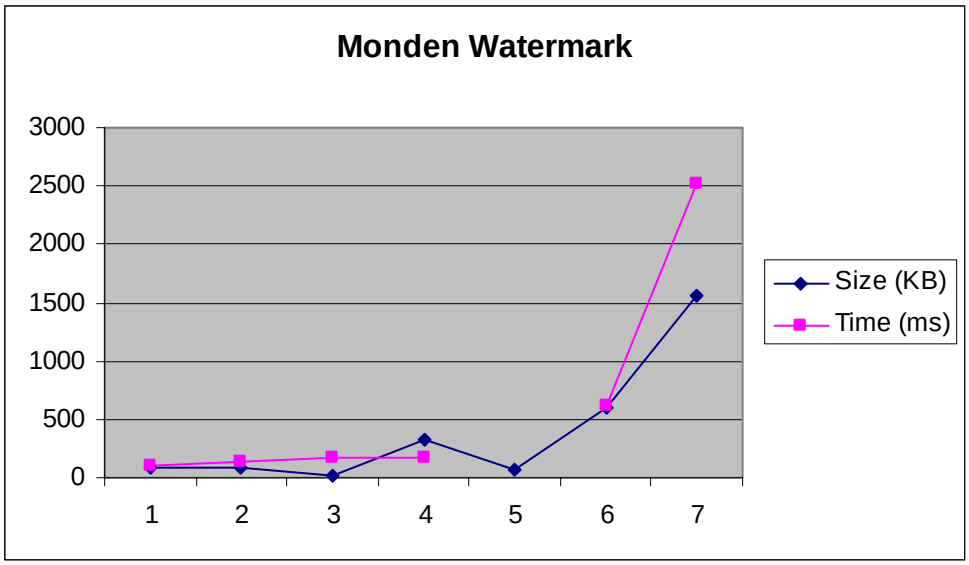

Figure 16: Size over time ratio with Monden watermark

Graph Theoretic Watermark

No	**Application**	**Size(KB)**	**Time(s)**	**Time(ms)**
1	spiro	115	0.924	92.4
2	jdrill2_3_1	x	x	x
3	TTT	x	x	x
4	Cvt2Mae	x	x	x
5	XMLTree	91.4	2.282	228.2
6	toy_1.4	x	x	x
7	Conzilla1.1Beta2	x	x	x
	Average	1.603	103.2	160.3

Table 35: Size over time ratio with GTW watermark

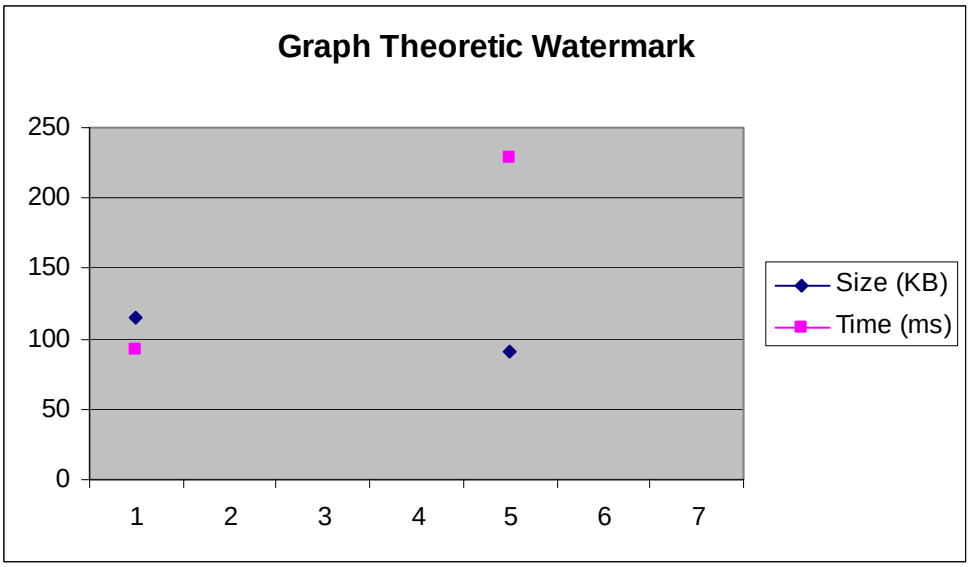

Figure 17: Size over time ratio with GTW watermark

AddMethField

No	**Application**	**Size(KB)**	**Time(s)**	**Time(ms)**
1	spiro	81.3	0.904	90.4
2	jdrill2_3_1	85.6	1.392	139.2
3	TTT	9.32	1.644	164.4
4	Cvt2Mae	326	1.572	157.2
5	XMLTree	62.3	2.188	218.8
6	toy_1.4	573	5.628	562.8
7	Conzilla1.1Beta2	1566	24.86	2486
	Average	5.455429	386.2171	545.5429

Table 36: Size over time ratio with AddMethodField watermark

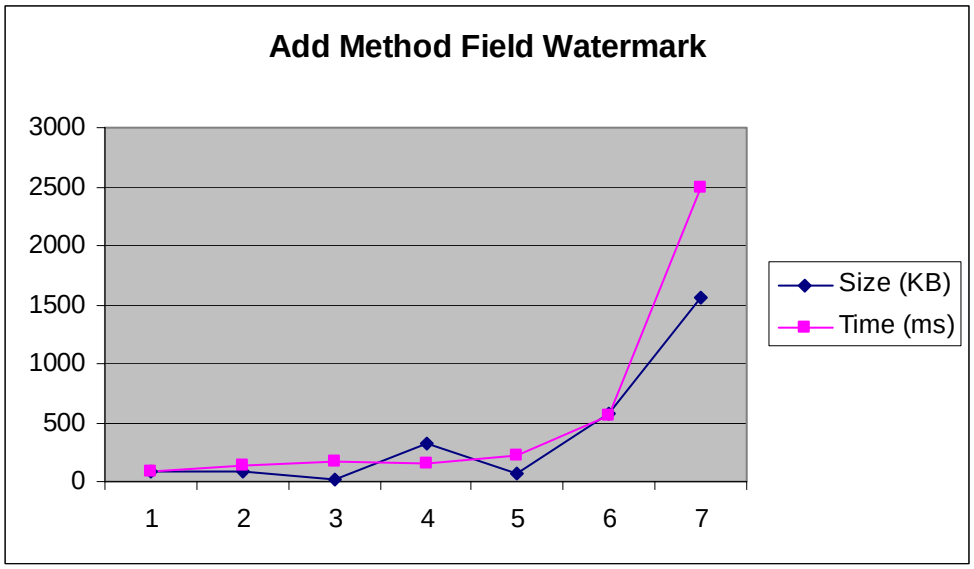

Figure 18: Size over time ratio with AddMethodField watermark

- The variation between size and time could vary depending on the application itself.

XMLTree	Size (KB)	Time (s)	Time (ms)
No Watermark	57.2	2.126	212.6
String Constant	62.2	2.144	214.4
Stern	62.5	2.168	216.8
Register Type	63.6	2.158	215.8
QuPotkonjak	62	2.162	216.2
Monden	62.8	x	X
Graph Theoretic Watermarking	91.4	2.282	228.2
AddMethField	62.3	2.188	218.8
Add Switch	62.3	2.186	218.6
Add Initialization	62.2	2.058	205.8
Add Expression	62.2	2.222	222.2
Average	64.60909	2.1694	216.94

Table 37: Size over time ratio for XMLTree Application

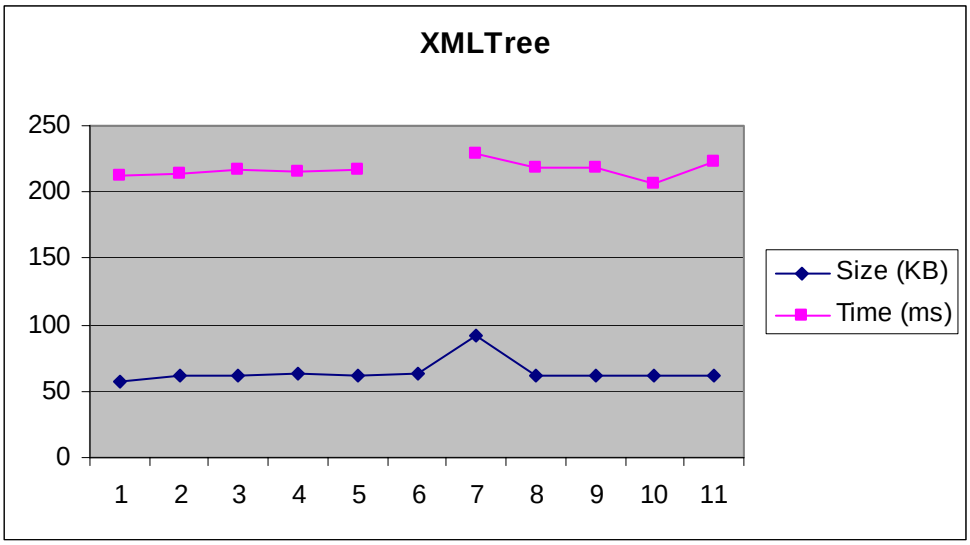

Figure 19: Size over time ratio for XMLTree Application

XMLTree	Size (%)	Time (%)
No Watermark	0	0
String Constant	11.76	0.84
Stern	12.47	1.94
Register Type	15.06	1.48
QuPotkonjak	11.29	1.67
Monden	13.18	x
Graph Theoretic Watermarking	37.42	6.84
AddMethField	12	2.83
Add Switch	12	2.74
Add Initialization	11.76	3.3
Add Expression	11.76	4.32
Average	13.51818	2.596

Table 38: Percentage of Size over time ratio for XMLTree Application

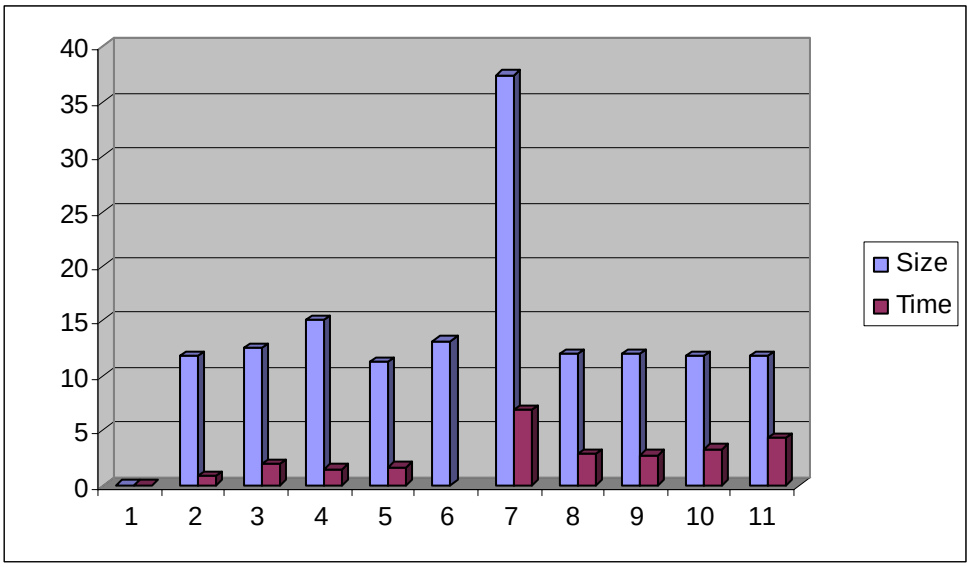

Figure 20: Percentage of Size over time ratio for XMLTree Application

spiro	**Size (KB)**	**Time (s)**	**Time (ms)**
No Watermark	73.5	0.801	80.1
String Constant	81.1	0.868	86.8
Stern	81.4	0.906	90.6
Register Type	82.1	0.871	87.1
QuPotkonjak	80.5	0.861	86.1
Monden	85	0.945	94.5
Graph Theoretic Watermarking	115	0.924	92.4
AddMethField	81.3	0.904	90.4
Add Switch	81.3	0.925	92.5
Add Initialization	81.2	0.896	89.6
Add Expression	81.2	0.889	88.9
Average	83.96364	0.89	89

Table 39: Size over time ratio for spiro Application

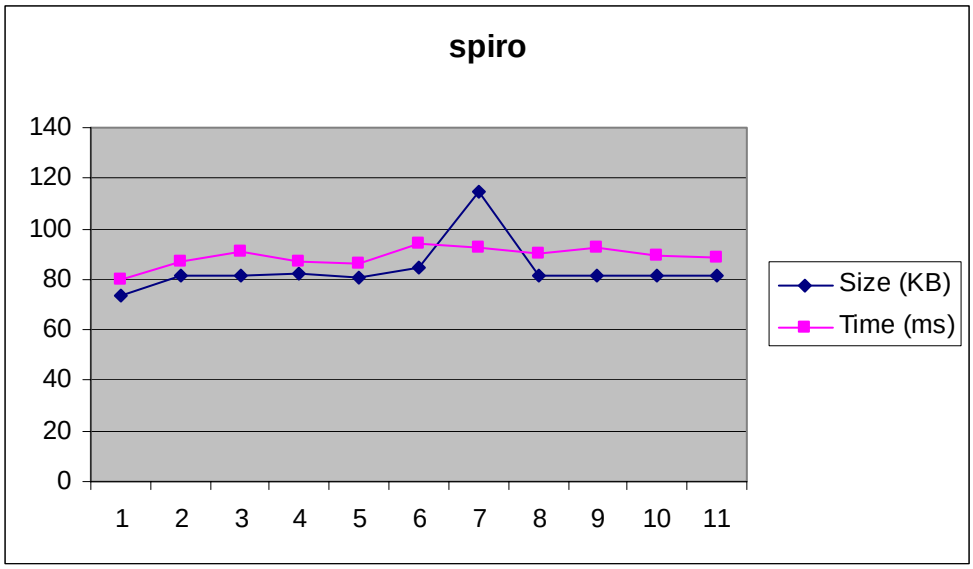

Figure 21: Size over time ratio for spiro Application

spiro	Size (%)	Time (%)
No Watermark	0	0
String Constant	17.88	7.72
Stern	18.59	11.59
Register Type	20.24	8.04
QuPotkonjak	16.47	6.97
Monden	27.06	15.24
Graph Theoretic Watermarking	97.65	13.31
AddMethField	18.35	11.39
Add Switch	18.35	13.41
Add Initialization	18.12	10.6
Add Expression	18.12	9.9
Average	24.62091	9.833636

Table 40: Percentage of Size over time ratio for spiro Application

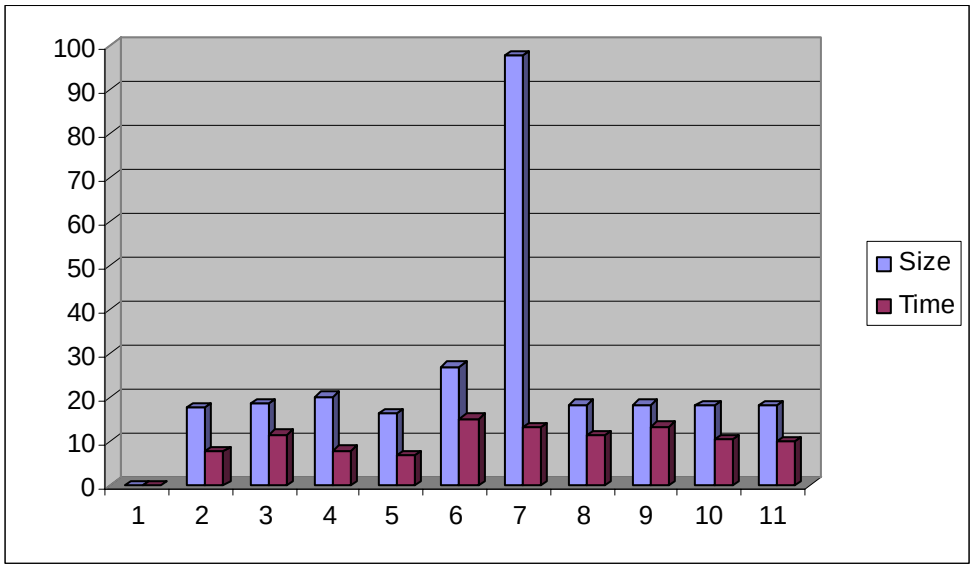

Figure 22: Percentage of Size over time ratio for spiro Application

From the previous analysis, a possible benchmarking factor can be added for analyzing the watermark class with reference to size and time expansions. The value is measured by dividing the ratio of size expansion over the ration of time expansion for a certain watermarking scheme.

	Level 1	Level 2	Level 3	Level 4	Level 5
Size over Time Expansion	< 5 %	5 – 10 %	10 – 20 %	20 – 50 %	> 50 %

Table 41: Classification based on Size over Time Expansion

7.4.3 Obfuscation Attacks

The results also showed that some obfuscation attacks could destroy the watermark completely while some do not have any perceptible effect. We can

conclude that some watermarking techniques are vulnerable to some obfuscation attacks regardless of the application being watermarked. Therefore, if an obfuscation attack breaks one watermarking scheme, then the same will be the case if it applied to other applications. The opposite is also true. Applying any obfuscation attack into an application could destroy the application and make it no more executable. Otherwise, the result will be simply either that W is recognizable or it is not.

	Class 1	Class 2	Class 3
Obfuscation attack	Execution stopped	W can be recognized	W can not be recognized

Table 42: Classification based on Obfuscation attack

7.4.4 Optimization Attacks

Optimization attacks either destroy the watermark or not regardless of the application being watermarked. Only two watermarking schemes out of the ten being experimented showed variation on the effect of the optimization attack depending on the tested application itself. Evaluating the effect of an optimization attack will result on one of the three possible cases: W will be always recognized, W will be recognized in some applications but no all, and W will never be recognized.

	Class 1	Class 2	Class 3
Optimization attack	W always recognized	W sometimes recognized	W never recognized

Table 43: Classification based on Optimization attack

7.4.5 Collusive Attacks

Results showed that collusive attacks are of great threat to watermarks, especially if the same watermarking scheme is applied twice. However, there is no clear trend on the relationship between the application and the different watermarks applied with reference to collusive attacks. The collusive attack could be by either applying the same watermark scheme used if known or by using another scheme. As in optimization attacks, applying collusive attacks could have three results as in the table below.

	Class 1	Class 2	Class 3
Adding same Watermarking scheme	W always recognized	W sometimes recognized	W never recognized
Adding another Watermarking scheme	W always recognized	W sometimes recognized	W never recognized

Table 44: Classification based on Collusive attacks

7.4.6 Manual Attacks

In manual attacks, if the watermark is easily found then it is highly probably that it will be easily removed as well. The opposite is most likely to be true. Evaluating this type of attacks seems to be vague. Therefore, the best way known to us for measuring such human-factor based attack is to look for individuals' evaluation as the table below suggests.

	Level 1	Level 2	Level 3	Level 4	Level 5
Finding W	V. Low	Low	Moderate	High	V. High
Removing W	V. Low	Low	Moderate	High	V. High

Table 45: Classification based on Manual attacks

In doing evaluation experimentations using the proposed benchmarking above, it is expected to repeated the tests several times and to use several types of attacks as well. Evaluation level will surely depend on the number and types of attacks being used and the applications where watermarks are embedded on. An example of an evaluation outline based on the proposed benchmarking is shown in the table below.

	Level 1	Level 2	Level 3	Level 4	Level 5
Expansion in Size	< 10 %	10 – 20 %	20 – 50 %	50 – 100 %	> 100 %
Expansion in Time	< 1 %	1 – 5 %	5 – 10 %	10 – 20 %	> 20 %
Size over Time Expansion	< 5 %	5 – 10 %	10 – 20 %	20 – 50 %	> 50 %

	Class 1	Class 2	Class 3
Obfuscation attack	Execution stopped	W can be recognized	W can not be recognized
Optimization attack	W always recognized	W sometimes recognized	W never recognized
Adding same Watermarking scheme	W always recognized	W sometimes recognized	W never recognized
Adding another Watermarking scheme	W always recognized	W sometimes recognized	W never recognized

Finding W	V. Low	Low	Moderate	High	V. High
Removing W	V. Low	Low	Moderate	High	V. High

Table 46: Summary of proposed benchmarking scheme for SW

CHAPTER 8

CONCLUSION

8.1 Findings and Results

In this thesis, several issues related to software watermarking were reviewed and studied. The matter of protecting copyrights and preventing piracy is very important and of great interest to industry and academic research. However, it was found that though it has many difficulties, working on the area of software security has many open areas for improvement. After discussing software protection and the major drivers for software piracy, an in-depth research on current software watermarking scheme was conducted.

Chapters 2 discussed the importance of software copyrights and ownership authentication. In Chapter 3, different security techniques such as hardware and software techniques where mentioned. The following chapter explained the reason of studying watermarking with java in this thesis. Next, a comparative study on different watermarking algorithms, including static and dynamic watermarks, was presented. Also, several possible attacks on software watermarks were analyzed.

Our major contribution to the software watermarking community in this thesis work was done in chapters 6 and 7. Chapter 6 was about an overview of the proposed work. First,

the need for evaluation benchmarking and some reminiscent unrelated work was presented. Then, our study approach was enlightened. After that, the proposed benchmarking factors were explained in details. In chapter 7, experimentation results, analysis, and observation were presented. Contributions and results of this thesis are summarized in the next section. In the last section, the limitations of the work done and future work were addressed.

8.2 Summary of Contributions

The outcomes and contributions of the work done on this thesis research can be summarized on the points below:

- Proposed, in details, a set of evaluation benchmarking attributes for any software watermarking scheme.
- Demonstrated the significance of the proposed approach through numerical experimentations comparing different varying sampling applications.
- Studied and surveyed the current and promising new techniques designed to reliably preserve and protect software programs.
- Elaborated on the state of art and promising future interests for research in the area of software watermarking.
- Evaluated and classified different software watermarking techniques and attacks.
- Showed that all existing schemes, even in future, are not immune to all types of attacks.

- Concluded that proposed dynamic watermarking schemes are vulnerable even to attacks designed for static watermarking schemes.
- Identified the need of having benchmarking tools for software watermarking to evaluate new schemes or to compare between existing schemes, which will speed up the research in this area.
- Presented extensive experimentations and results of different watermarks and attacks.
- Identified some promising new areas of improvements and future work on the area of software watermarking.

8.3 Limitations and Future Work

The limitations of the work done in this thesis are as follows:

- Only few references and work were done in this area since the existing research on software watermarking is considered new.
- Many proposed software watermarks are not completely explained or not even implemented.
- Very limited number of software watermarking tools available, and all are in evaluation (beta) versions.
- In the experimentations, only watermarks that are applied to bytecode and not source code were considered. Moreover, the proposed benchmarking assumes that there is no partial recognition of watermarks (non-binary problem).

- Evaluating manual attacks is still lacking quality metrics that do not depend heavily on human factors.

After the work done in this thesis, many opening areas for future work were identified. First of all, the limitations listed above are to be addressed. In addition, more watermarking schemes can be considered to fatherly improve the benchmarking. Moreover, we will try to find the feasibility of having automatic benchmarking tool for software watermarking schemes. Another possibility is to work on developing a tool that utilizes the information obtained from the proposed benchmarking in identifying the watermark scheme being applied into an application.

Testing the benchmark with newly created software watermarking algorithms is also considered in future work. Having results calculated for such testing will help in coming up with a possible new ideas in this area. And finally, applying watermarking on programs written on code other than Java, such as C#, will be another objective of our future research.

REFERENCES

[1] B. Anckaert, Bjorn De Sutter and Koen De Bosschere. Software Piracy Prevention through Diversity. Proceedings of the 4th ACM workshop on Digital rights management, p.63-71, 2004.

[2] C. Collberg and C. Thomborson, On the limits of Software Watermarking, Technical Report #164, Department of Computer Science, The University of Auckland, August 1998.

[3] C. Collberg and C. Thomborson, (1999), Software watermarking: Models and dynamic embeddings, in `Principles of Programming Languages (POPL'99)', San Antonio, TX, pp. 311-324.

[4] C. Collberg, Clark Thomborson, and Gregg Townsend. Dynamic graph-based software watermarking. Technical report, Dept. of Computer Science, Univ. of Arizona, 2004.

[5] C. Collberg, Andrew Huntwork, Edward Carter, and Gregg Townsend. Graph Theoretic Software Watermarks: Implementation, Analysis, and Attacks. In 6th Information Hiding Workshop, 2004.

[6] C. Collberg, E. Carter, S. Debray, A. Huntwork, C. Linn, M. Stepp, Dynamic Path-Based Software Watermarking. PLDI 2004.

[7] P. Cousot and Radhia Cousot.. An Abstract Interpretation-Based Framework for Software Watermarking. In Conference Record of the 31st ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Programming Languages, Venice, Italy, January 14-16, 2004. ACM Press, New York, U.S.A. pp. 173—185.

[8] D. Curran, N.J. Hurley, and M. O. Cinneide. Securing java through software watermarking. In *Proceedings of the 2nd international conference on Principles and practice of programming in Java*, 2003.

[9] Davidson; Robert I. (Bellevue, WA);Myhrvold; Nathan (Bellevue, WA), Microsoft Corporation (Redmond, WA), Computer software protection. United States Patent 5559884. 1996. http://freepatentsonline.com/5559884.html

[10] K. Holmes; (Dublin, IE), International Business Machines Corporation (Armonk, NY), Computer software protection. United States Patent 5287407. 1994. http://freepatentsonline.com/5287407.html

[11] Erin Joyce "Software Piracy Losses Add Up to $29B." July 8, 2004. http://www.enterpriseitplanet.com/security/news/article.php/3378251

[12] M. Kutter and F. A. Petitcolas "*A Fair Benchmark for Image Watermarking Systems*" Proc. SPIE Security and Watermarking of Multimedia Contents vol. 3657, pp. 226-239, January 1999.

[13] Moskowitz; Scott A. (North Miami Beach, FL);Cooperman; Marc (Palo Alto, CA), The Dice Company (Miami, FL), Computer software protection. United States Patent 5745569. 1998. http://freepatentsonline.com/5745569.html

[14] J. Nagra, C. Thomborson, and C. Collberg, (2002), A functional taxonomy for software watermark- ing, in M. Oudshoorn, ed., `Proc. 25th Australasian Computer Science Conference 2002', ACS, pp. 177-186.

[15] J. Palsberg and S. Krishnaswami, Kwon, M., Ma, D., Shao, Q. & Zhang, Y. (2001), Experience with software watermarking, in `Proc. 16th Ann. Comp. Security Applications Conf. (AC-SAC'00)', IEEE Computer Society, pp. 308-316.

[16] fabien a. p. petitcolas http://www.petitcolas.net/fabien/watermarking/stirmark/

[17] FAP Petitcolas,"Watermarking Schemes Evaluation", IEEE Magazine of Signal. Processing, Vol. 17, No. 5, , pp. 58-64, September 2000.

[18] T. Premkumar Devanbu and Stuart Stubblebine. Software engineering for security: a roadmap. In *Proceedings of the conference on The future of Software engineering*, pages 227-239. ACM Press, 2000.

[19] P. R. Samson; (San Francisco, CA), Autodesk, Inc. (Sausalito, CA), Computer software protection. United States Patent 5287408. 1994. http://freepatentsonline.com/5287408.html

[20] C. Thomborson, Nagra, J., Somaraju, R. and He, C. (2004). Tamper-proofing Software Watermarks. In Proc. Second Australasian Information Security Workshop (AISW2004), Dunedin, New Zealand. CRPIT, 32. Montague, P. and Steketee, C., Eds., ACS. 27-36.

[21] "One Third of All Software in Use Still Pirated, Major Study Finds." International Data Corporation (IDC). "http://www.idc.com/getdoc.jsp?containerId=prUS00150505". WASHINGTON, D.C., May 18, 2005.

[22] Business Software Alliance www.bsa.org.

[23] The Easter Egg Archive™ [www.eeggs.com/items/568.html]

[24] Watermarking - Quality Evaluations
http://wwwiti.cs.uni-magdeburg.de/iti_amsl/lehre/02_SoSem/mmsec/watermarking/

APPENDIX

Experimental Results

execution time (ms)

XMLTree.jar			
No Watermark			
#	**S**	**E**	**S-E**
1	45.71	47.74	2.03
2	6.1	8.29	2.19
3	24.45	26.61	2.16
4	48.47	50.54	2.07
5	7.6	9.78	2.18
average			**2.126**
stdv			**0.072**

Monden			
#	**S**	**E**	**S-E**
1			0
2			0
3			0
4			0
5			0
average			**0**
stdv			**0**

AddExpression			
#	**S**	**E**	**S-E**
1	40.01	42.24	2.23
2	26.36	28.62	2.26
3	43.5	45.7	2.2
4	59.47	61.62	2.15
5	18.47	20.74	2.27
average			**2.222**
stdv			**0.049**

String Constant			
#	**S**	**E**	**S-E**
1	20.9	23.07	2.17
2	12.64	14.77	2.13
3	44.21	46.39	2.18
4	52.99	55.06	2.07
5	8.92	11.09	2.17
average			**2.144**

stdv	**0.046**

GraphTheoreticWatermarking			
#	**S**	**E**	**S-E**
1	41.37	43.69	2.32
2	45.14	47.43	2.29
3	48.6	50.85	2.25
4	51.68	53.94	2.26
5	54.76	57.05	2.29
average			**2.282**
stdv			**0.028**

Qu Potkonjak			
#	**S**	**E**	**S-E**
1	35.3	37.38	2.08
2	58.48	60.65	2.17
3	19.69	21.93	2.24
4	33.75	35.81	2.06
5	52.63	54.89	2.26
average			**2.162**
stdv			**0.091**

Stern			
#	**S**	**E**	**S-E**
1	34.41	36.52	2.11
2	23.28	25.41	2.13
3	38.34	40.65	2.31
4	52.37	54.44	2.07
5	11.28	13.5	2.22
average			**2.168**
stdv			**0.097**

AddSwitch			
#	**S**	**E**	**S-E**
1	34.49	36.62	2.13
2	4.69	6.83	2.14
3	7.32	9.54	2.22
4	6.05	8.17	2.12
5	21.09	23.41	2.32
average			**2.186**

stdv	0.085

AddInitialization			
#	S	E	S-E
1	3.04	5.12	2.08
2	31.59	33.56	1.97
3	44.33	46.3	1.97
4	55.87	58.04	2.17
5	9.99	12.09	2.1
average			2.058
stdv			0.087

RegisterType			
#	S	E	S-E
1	59.84	61.97	2.13
2	23.79	25.85	2.06
3	39.89	42.11	2.22
4	20.65	22.71	2.06
5	6.84	9.16	2.32
average			2.158
stdv			0.112

AddMethField			
#	S	E	S-E
1	42.24	44.48	2.24
2	43.29	45.57	2.28
3	28.16	30.24	2.08
4	52.94	55.02	2.08
5	31.21	33.47	2.26
average			2.188
stdv			0.1

execution time (ms)

spiro.jar			
No Watermark			
#	S	E	S-E
1	48.1	49.01	0.91
2	23.07	23.86	0.79
3	52.01	52.82	0.81
4	26.11	26.95	0.84
5	41.91	42.7	0.79
6	55.74	56.53	0.79
7	18.39	19.08	0.69
8	35.96	36.71	0.75
9	51.14	51.93	0.79
10	6.56	7.41	0.85
average			0.801
stdv			0.059

Monden			
#	S	E	S-E
1	12.97	13.85	0.88
2	39.44	40.42	0.98
3	56.79	57.68	0.89
4	14.09	15.03	0.94
5	31.45	32.43	0.98
6	26.93	27.92	0.99
7	15.76	16.76	1
8	14.14	15.11	0.97
9	31.71	32.64	0.93
10	39.32	40.21	0.89
average			0.945
stdv			0.046

String Constant			
#	S	E	S-E
1	38.92	39.77	0.85
2	50.58	51.44	0.86
3	18.59	19.4	0.81
4	36.7	37.59	0.89
5	51.55	52.36	0.81
6	37.81	38.66	0.85
7	52.59	53.44	0.85
8	5.36	6.31	0.95
9	21.95	22.9	0.95
10	36.36	37.22	0.86
average			0.868
stdv			0.049

GraphTheoreticWatermarking			
#	S	E	S-E
1	40.21	41.14	0.93
2	35.51	36.37	0.86
3	55.09	55.95	0.86
4	11.7	12.61	0.91
5	7.83	8.74	0.91
6	41.35	42.35	1
7	17.48	18.4	0.92
8	28.39	29.36	0.97
9	35.08	36.05	0.97

10	24.34	25.25	0.91
average			**0.924**
stdv			**0.046**

Stern			
#	**S**	**E**	**S-E**
1	29.93	30.89	0.96
2	41.72	42.58	0.86
3	6.63	7.53	0.9
4	29.35	30.25	0.9
5	39.5	40.41	0.91
6	51.43	52.38	0.95
7	55.36	56.22	0.86
8	23.05	24.04	0.99
9	41.14	42.01	0.87
10	30.21	31.07	0.86
average			**0.906**
stdv			**0.047**

AddSwitch			
#	**S**	**E**	**S-E**
1	48.09	48.97	0.88
2	13.75	14.66	0.91
3	24.89	25.81	0.92
4	49.46	50.41	0.95
5	0.98	1.85	0.87
6	13.88	14.84	0.96
7	14.39	15.29	0.9
8	38.53	39.51	0.98
9	50.42	51.29	0.87
10	5.16	6.17	1.01
average			**0.925**
stdv			**0.048**

RegisterType			
#	**S**	**E**	**S-E**
1	44.86	45.75	0.89
2	4.63	5.53	0.9
3	32.77	33.58	0.81
4	46.96	47.86	0.9
5	24.76	25.62	0.86
6	14.99	15.79	0.8
7	28.57	29.43	0.86
8	41.47	42.3	0.83
9	7.64	8.59	0.95
10	25.86	26.77	0.91
average			**0.871**
stdv			**0.048**

AddMethField			
#	**S**	**E**	**S-E**
1	12.35	13.26	0.91
2	44.38	45.29	0.91
3	56.49	57.39	0.9
4	10.02	10.92	0.9
5	23.07	24.03	0.96
6	37.62	38.52	0.9
7	51.73	52.58	0.85
8	4.56	5.4	0.84
9	17.25	18.12	0.87
10	29.12	30.12	1
average			**0.904**
stdv			**0.048**
Qu Potkonjak			
#	**S**	**E**	**S-E**
1	6.04	6.94	0.9
2	20.24	21.15	0.91
3	49.29	50.24	0.95
4	26.73	27.56	0.83
5	18.1	18.91	0.81
6	31.06	31.87	0.81
7	45.05	45.88	0.83
8	1.22	2.05	0.83
9	5.16	6.05	0.89
10	31.63	32.48	0.85
average			**0.861**
stdv			**0.048**

AddInitialization			
#	**S**	**E**	**S-E**
1	19.75	20.65	0.9
2	17.1	17.92	0.82
3	55.92	56.83	0.91
4	16.24	17.14	0.9
5	29.38	30.22	0.84
6	9.06	9.95	0.89
7	55.56	56.52	0.96
8	45.38	46.24	0.86
9	17.57	18.48	0.91
10	43.53	44.5	0.97
average			**0.896**
stdv			**0.047**

AddExpression			
#	S	E	S-E
1	4.91	5.82	0.91
2	19.73	20.64	0.91
3	36.58	37.48	0.9
4	51.95	52.88	0.93
5	6.34	7.25	0.91
6	20.49	21.44	0.95
7	32.59	33.39	0.8
8	45.07	45.88	0.81
9	58.43	59.34	0.91
10	12.93	13.79	0.86
average			**0.889**
stdv			**0.05**

execution time (ms)

toy_1.4.jar			
No Watermark			
#	S	E	S-E
1	29.94	35.38	5.44
2	34.08	39.53	5.45
3	17.51	22.96	5.45
4	48.18	53.62	5.44
5	54.75	60.25	5.5
average			**5.456**
stdv			**0.025**

Monden			
#	S	E	S-E
1	12.6	19.08	6.48
2	44.97	50.58	5.61
3	54.32	61.05	6.73
4	2.74	8.8	6.06
5	29.53	35.25	5.72
average			**6.12**
stdv			**0.481**

Qu Potkonjak			
#	S	E	S-E
1	58.29	65.18	6.89
2	5.9	11.27	5.37
3	12.25	17.71	5.46
4	43.36	48.74	5.38
5	50.35	56.25	5.9
average			**5.8**

stdv	**0.647**

String Constant			
#	S	E	S-E
1	24.96	30.55	5.59
2	32.13	37.94	5.81
3	38.86	44.35	5.49
4	45.13	50.57	5.44
5	31.37	36.89	5.52
average			**5.57**
stdv			**0.145**

GraphTheoreticWatermarking			
#	S	E	S-E
1			0
2			0
3			0
4			0
5			0
average			**0**
stdv			**0**

AddInitialization			
#	S	E	S-E
1	42.89	48.32	5.43
2	32.56	38.19	5.63
3	49.74	55.26	5.52
4	5.97	11.58	5.61
5	21.03	26.55	5.52
average			**5.542**
stdv			**0.08**

Stern			
#	S	E	S-E
1	40.14	45.67	5.53
2	34.2	39.71	5.51
3	27.77	33.12	5.35
4	21.73	27.29	5.56
5	36.06	42.2	6.14
average			**5.618**
stdv			**0.303**

AddSwitch			
#	S	E	S-E
1	29.67	34.88	5.21
2	21.04	26.61	5.57

3	36.87	42.49	5.62
4	54.37	59.9	5.53
5	1.37	6.98	5.61
average			**5.508**
stdv			**0.17**

AddExpression			
#	**S**	**E**	**S-E**
1	51.25	57.42	6.17
2	58.39	64.44	6.05
3	30.42	36.47	6.05
4	43.31	48.63	5.32
5	48.84	54.36	5.52
average			**5.822**
stdv			**0.377**

RegisterType			
#	**S**	**E**	**S-E**
1	37.41	43.05	5.64
2	43.79	49.05	5.26
3	50.59	55.99	5.4
4	56.89	62.18	5.29
5	33.43	38.99	5.56
average			**5.43**
stdv			**0.166**

AddMethField			
#	**S**	**E**	**S-E**
1	38.27	44.1	5.83
2	45.84	51.31	5.47
3	18.53	24.22	5.69
4	26.49	32.12	5.63
5	16.6	22.12	5.52
average			**5.628**
stdv			**0.143**

TTT.jar			
No Watermark			
#	**S**	**E**	**S-E**
1	26.31	27.9	1.59
2	29.69	31.24	1.55
3	4.54	6.18	1.64
4	54.71	56.36	1.65
5	11.33	13.02	1.69
average			**1.624**
stdv			**0.055**

Monden			
#	**S**	**E**	**S-E**
1	27.34	28.87	1.53
2	42.57	44.28	1.71
3	55.68	57.33	1.65
4	8.21	9.86	1.65
5	21.3	23	1.7
average			**1.648**
stdv			**0.072**

Qu Potkonjak			
#	**S**	**E**	**S-E**
1	59.02	60.77	1.75
2	20.56	22.27	1.71
3	35.63	37.32	1.69
4	49.66	51.25	1.59
5	1.32	2.97	1.65
average			**1.678**
stdv			**0.061**

String Constant			
#	**S**	**E**	**S-E**
1	53.34	55.15	1.81
2	9.68	11.35	1.67
3	22.16	23.82	1.66
4	38.13	39.78	1.65
5	53.96	55.6	1.64
average			**1.686**
stdv			**0.07**

GraphTheoreticWatermarking			
#	**S**	**E**	**S-E**
1			0
2			0
3			0
4			0
5			0
average			**0**
stdv			**0**

AddInitialization			
#	**S**	**E**	**S-E**
1	28.26	29.98	1.72
2	43.37	44.94	1.57
3	55.19	56.76	1.57
4	7.64	9.21	1.57

5	18.84	20.42	1.58
average			**1.602**
stdv			**0.066**

Stern			
#	**S**	**E**	**S-E**
1	41.95	43.67	1.72
2	1.31	3.12	1.81
3	15.31	17.01	1.7
4	33.23	34.82	1.59
5	44.56	46.15	1.59
average			**1.682**
stdv			**0.094**

AddSwitch			
#	**S**	**E**	**S-E**
1	31.43	33.12	1.69
2	54.51	56.25	1.74
3	10.2	11.85	1.65
4	49.2	50.79	1.59
5	2.18	3.76	1.58
average			**1.65**
stdv			**0.067**

AddExpression			
#	**S**	**E**	**S-E**
1	51.02	52.64	1.62
2	26.71	28.32	1.61
3	18.01	19.65	1.64
4	30.52	32.17	1.65
5	44.33	45.94	1.61
average			**1.626**
stdv			**0.018**

RegisterType			
#	**S**	**E**	**S-E**
1	34.28	35.92	1.64
2	57.64	59.38	1.74
3	2.51	4.19	1.68
4	21.89	23.53	1.64
5	35.45	37.2	1.75
average			**1.69**
stdv			**0.053**

AddMethField			
#	**S**	**E**	**S-E**
1	17.3	19.03	1.73
2	35.87	37.48	1.61
3	50.21	51.87	1.66
4	2.9	4.48	1.58
5	12.47	14.11	1.64
average			**1.644**
stdv			**0.057**

jdrill2_3_1.jar			
No Watermark			
#	**S**	**E**	**S-E**
1	28.15	29.59	1.44
2	19.59	20.98	1.39
3	22.22	23.66	1.44
4	5.72	7.15	1.43
5	3.49	4.94	1.45
average			**1.43**
stdv			**0.023**

Monden			
#	**S**	**E**	**S-E**
1	10.04	11.5	1.46
2	12.1	13.62	1.52
3	14.16	15.5	1.34
4	16.06	17.55	1.49
5	45.98	47.36	1.38
average			**1.438**
stdv			**0.076**

Qu Potkonjak			
#	**S**	**E**	**S-E**
1	45.45	46.89	1.44
2	50.33	51.83	1.5
3	52.76	54.15	1.39
4	54.73	56.07	1.34
5	43.17	44.66	1.49
average			**1.432**
stdv			**0.068**

String Constant			
#	**S**	**E**	**S-E**
1	35.3	36.85	1.55
2	37.68	39.08	1.4
3	46.02	47.3	1.28
4	41.8	43.25	1.45
5	43.93	45.33	1.4
average			**1.416**

stdv	0.098

GraphTheoreticWatermarking			
#	S	E	S-E
1			0
2			0
3			0
4			0
5			0
average			0
stdv			0

AddInitialization			
#	S	E	S-E
1	21.15	22.47	1.32
2	36.47	37.88	1.41
3	38.43	39.83	1.4
4	16.57	17.99	1.42
5	18.7	20.16	1.46
average			1.402
stdv			0.051

Stern			
#	S	E	S-E
1	28.89	30.19	1.3
2	16.72	18.08	1.36
3	48.52	49.88	1.36
4	46.26	47.66	1.4
5	44.25	45.7	1.45
average			1.374
stdv			0.055

AddSwitch			
#	S	E	S-E
1	26.66	28.1	1.44
2	48.77	50.02	1.25
3	50.52	51.97	1.45
4	52.5	53.8	1.3
5	28.71	30.05	1.34
average			1.356
stdv			0.087

AddExpression			
#	S	E	S-E
1	56.95	58.3	1.35
2	54.96	56.4	1.44
3	15.65	17.1	1.45
4	17.65	18.96	1.31
5	52.91	54.36	1.45
average			1.4
stdv			0.066

RegisterType			
#	S	E	S-E
1	17.4	18.79	1.39
2	19.92	21.28	1.36
3	22	23.26	1.26
4	24.03	25.44	1.41
5	56.41	57.75	1.34
average			1.352
stdv			0.058

AddMethField			
#	S	E	S-E
1	13.15	14.49	1.34
2	44.17	45.65	1.48
3	46.3	47.6	1.3
4	17.17	18.52	1.35
5	15.13	16.62	1.49
average			1.392
stdv			0.087
Cvt2Mae.jar			
No Watermark			
#	S	E	S-E
1	45.17	46.82	1.65
2	28.58	30.23	1.65
3	49.1	50.74	1.64
4	51.16	52.71	1.55
5	30.5	32.16	1.66
average			1.63
stdv			0.045

Monden			
#	S	E	S-E
1	49.46	51.13	1.67
2	40.92	42.53	1.61
3	42.94	44.6	1.66
4	51.56	53.29	1.73
5	47.29	48.99	1.7
average			1.674
stdv			0.045

Qu Potkonjak			
#	S	E	S-E
1			0
2			0
3			0
4			0
5			0
average			**0**
stdv			**0**

String Constant			
#	S	E	S-E
1	17.98	19.64	1.66
2	19.98	21.64	1.66
3	21.98	23.63	1.65
4	23.97	25.63	1.66
5	26.01	27.66	1.65
average			**1.656**
stdv			**0.005**

GraphTheoreticWatermarking			
#	S	E	S-E
1			0
2			0
3			0
4			0
5			0
average			**0**
Stdv			**0**

AddInitialization			
#	S	E	S-E
1	23.27	24.83	1.56
2	33.04	34.7	1.66
3	17.36	18.93	1.57
4	19.62	21.38	1.76
5	3.31	4.94	1.63
average			**1.636**
stdv			**0.081**

Stern			
#	S	E	S-E
1	7.61	9.26	1.65
2	5.6	7.21	1.61
3	57.01	58.71	1.7
4	1.49	3.2	1.71
5	59.23	60.99	1.76
average			**1.686**
stdv			**0.058**

AddSwitch			
#	S	E	S-E
1	50.19	51.81	1.62
2	42.79	44.38	1.59
3	44.79	46.33	1.54
4	47.03	48.57	1.54
5	32.62	34.27	1.65
average			**1.588**
stdv			**0.049**

AddExpression			
#	S	E	S-E
1	3.96	5.6	1.64
2	6.04	7.69	1.65
3	8.03	9.74	1.71
4	47.74	49.44	1.7
5	50.36	52.06	1.7
average			**1.68**
stdv			**0.032**

RegisterType			
#	S	E	S-E
1	11.6	13.27	1.67
2	13.67	15.37	1.7
3	15.74	17.51	1.77
4	18.02	19.68	1.66
5	21.87	23.57	1.7
average			**1.7**
stdv			**0.043**

AddMethField			
#	S	E	S-E
1	21.88	23.54	1.66
2	24.15	25.69	1.54
3	41.27	42.9	1.63
4	28.38	29.87	1.49
5	21.53	23.07	1.54
average			**1.572**
stdv			**0.07**
Conzilla1.1Beta2.jar			
No Watermark			
#	S	E	S-E

1	9.91	35.16	25.25
2	37.85	62.91	25.06
3	5.13	29.73	24.6
4	11.41	36.91	25.5
5	49.77	74.97	25.2
average			**25.12**
stdv			**0.332**

Monden			
#	**S**	**E**	**S-E**
1	6.26	31.18	24.92
2	33.03	58.5	25.47
3	0.76	25.62	24.86
4	31.46	56.48	25.02
5	59.66	85.11	25.45
average			**25.14**
stdv			**0.294**

Qu Potkonjak			
#	**S**	**E**	**S-E**
1	55.42	81.8	26.38
2	23.93	48.98	25.05
3	57.87	82.76	24.89
4	37.16	61.8	24.64
5	15	40.19	25.19
average			**25.23**
stdv			**0.675**

String Constant			
#	**S**	**E**	**S-E**
1	10.2	36.24	26.04
2	44.01	69.01	25
3	22.64	46.67	24.03
4	50.7	75.49	24.79
5	17.27	42.29	25.02
average			**24.98**
stdv			**0.718**

GraphTheoreticWatermarking			
#	**S**	**E**	**S-E**
1			0
2			0
3			0
4			0
5			0
average			**0**

stdv	**0**

AddInitialization			
#	**S**	**E**	**S-E**
1	26.32	53.09	26.77
2	54.33	79.1	24.77
3	20.7	45.28	24.58
4	46.57	71.68	25.11
5	13.14	37.52	24.38
average			**25.12**
stdv			**0.96**

Stern			
#	**S**	**E**	**S-E**
1	15.45	39.76	24.31
2	42.16	67.57	25.41
3	10.41	35.41	25
4	38.86	64.65	25.79
5	9.5	34.69	25.19
average			**25.14**
stdv			**0.549**

AddSwitch			
#	**S**	**E**	**S-E**
1	32	56.39	24.39
2	58.84	85.29	26.45
3	27.45	53.41	25.96
4	54.92	80.27	25.35
5	24.73	50.65	25.92
average			**25.61**
stdv			**0.787**

AddExpression			
#	**S**	**E**	**S-E**
1	37.66	62.48	24.82
2	16.11	41.55	25.44
3	48.82	75.16	26.34
4	19.69	44.97	25.28
5	48.41	73.41	25
average			**25.38**
stdv			**0.59**

RegisterType			
#	**S**	**E**	**S-E**
1	9.47	34.26	24.79
2	44.1	69.11	25.01

3	22.2	47.54	25.34
4	50.5	74.9	24.4
5	24.65	48.99	24.34
average			**24.78**
stdv			**0.42**

AddMethField			
#	**S**	**E**	**S-E**
1	31.07	55.84	24.77
2	22.42	47.11	24.69
3	48.61	73	24.39
4	14.54	39.88	25.34
5	43.69	68.81	25.12
average			**24.86**
stdv			**0.373**

Execution Time (ms)			Watermarked Execution Time (ms)									
Application		No W	**W1**	**W2**	**W3**	**W4**	**W5**	**W6**	**W 7**	**W8**	**W9**	**W10**
XMLTree	Avg	2.126	2.144	2.168	2.158	2.162		2.282	2.1 88	2.186	2.058	2.222
XMLTree	Stdv	0.072	0.046	0.097	0.112	0.091		0.028	0.1	0.085	0.087	0.049
TTT	Avg	1.624	1.686	1.682	1.69	1.678	1.648		1.6 44	1.65	1.602	1.626
TTT	Stdv	0.055	0.07	0.094	0.053	0.061	0.072		0.0 57	0.067	0.066	18
toy_1.4	Avg	5.456	5.57	5.618	5.43	5.8	6.12		5.6 28	5.508	5.542	5.822
toy_1.4	Stdv	0.025	0.145	0.303	0.166	0.647	0.481		0.1 43	0.17	0.08	0.377
spiro	Avg	0.801	0.868	0.906	0.871	0.861	0.945	0.924	0.9 04	0.925	0.896	0.889
spiro	Stdv	0.059	0.049	0.047	0.048	0.048	0.046	0.046	0.0 48	0.048	0.047	0.050
jdrill2_3_1	Avg	1.43	1.416	1.374	1.352	1.432	1.438		1.3 92	1.356	1.402	1.4
jdrill2_3_1	Stdv	0.023	0.098	0.055	0.058	0.068	0.076		0.0 87	0.087	0.051	0.066
Cvt2Mae	Avg	1.63	1.656	1.686	1.7		1.674		1.5 72	1.588	1.636	1.68
Cvt2Mae	Stdv	0.045	0.005	0.058	0.043		0.045		0.0 7	0.049	0.081	0.032
Conzilla1	Avg	25.12	24.98	25.14	24.78	25.23	25.14		24. 86	25.61	25.12	25.38
Conzilla1	Stdv	0.332	0.718	0.549	0.42	0.675	0.294		0.3 73	0.787	0.96	0.59

Execution Time (ms)			Difference in Watermarked Execution Time (ms)									
Application			**W1**	**W2**	**W3**	**W4**	**W5**	**W6**	**W7**	**W8**	**W9**	**W10**
XMLTree	Avg		0.018	0.042	0.03 2	0.03 6		0.156	0.062	0.06	- 0.068	0.096
XMLTree	Stdv		-0.026	0.025	0.04	0.01 9		-0.044	0.028	0.013	0.015	- 0.023
TTT	Avg		0.062	0.058	0.06 6	0.05 4	0.024		0.02	0.026	- 0.022	0.002
TTT	Stdv		0.015	0.039	- 0.00 2	0.00 6	0.017		0.002	0.012	0.011	17.94 5
toy_1.4	Avg		0.114	0.162	- 0.02 6	0.34 4	0.664		0.172	0.052	0.086	0.366
toy_1.4	Stdv		0.12	0.278	0.14 1	0.62 2	0.456		0.118	0.145	0.055	0.352

spiro	Avg		0.067	0.105	0.07	0.06	0.144	0.123	0.103	0.124	0.095	0.088
spiro	Stdv		-0.010	-0.012	-0.011	-0.011	-0.013	-0.013	-0.011	-0.011	-0.012	-0.009
jdrill2_3_1	Avg		-0.014	-0.056	-0.078	0.002	0.008		-0.038	-0.074	-0.028	-0.03
jdrill2_3_1	Stdv		0.075	0.032	0.035	0.045	0.053		0.064	0.064	0.028	0.043
Cvt2Mae	Avg		0.026	0.056	0.07		0.044		-0.058	-0.042	0.006	0.05
Cvt2Mae	Stdv		-0.04	0.013	-0.002		0		0.025	0.004	0.036	-0.013
Conzilla1	Avg		-0.14	0.02	-0.34	0.11	0.02		-0.26	0.49	0	0.26
Conzilla1	Stdv		0.386	0.217	0.088	0.343	-0.038		0.041	0.455	0.628	0.258

Execution Time (ms)			Change (%)									
Application			**W1**	**W2**	**W3**	**W4**	**W5**	**W6**	**W7**	**W8**	**W9**	**W10**
XMLTree	Avg		0.84	1.94	1.48	1.67		6.84	2.83	2.74	-3.30	4.32
XMLTree	Stdv		-56.52	25.77	35.71	20.88		-157.14	28.00	15.29	17.24	-46.94
TTT	Avg		3.68	3.45	3.91	3.22	1.46		1.22	1.58	-1.37	0.12
TTT	Stdv		21.43	41.49	-3.77	9.84	23.61		3.51	17.91	16.67	99.69
toy_1.4	Avg		2.05	2.88	-0.48	5.93	10.85		3.06	0.94	1.55	6.29
toy_1.4	Stdv		82.76	91.75	84.94	96.14	94.80		82.52	85.29	68.75	93.37
spiro	Avg		7.72	11.59	8.04	6.97	15.24	13.31	11.39	13.41	10.60	9.90
spiro	Stdv		-20.41	-25.53	-22.92	-22.92	-28.26	-28.26	-22.92	-22.92	-25.53	-18.00
jdrill2_3_1	Avg		-0.99	-4.08	-5.77	0.14	0.56		-2.73	-5.46	-2.00	-2.14
jdrill2_3_1	Stdv		76.53	58.18	60.34	66.18	69.74		73.56	73.56	54.90	65.15
Cvt2Mae	Avg		1.57	3.32	4.12		2.62843		-3.69	-2.64	0.37	2.98
Cvt2Mae	Stdv		-800.00	22.41	-4.65		0		35.71	8.16	44.44	-40.63
Conzilla1	Avg		-0.56	0.08	-1.37	0.44	0.08		-1.05	1.91	0.00	1.02
Conzilla1	Stdv		53.76	39.53	20.95	50.81	-12.93		10.99	57.81	65.42	43.73

VITA

- Mohannad Ahmad AbdulAziz Al-Dharrab

- Joined Saudi Aramco college preparatory program (CPP) - one year preparation program - in 1998 after finishing high school and graduated with first honor in September 1999.

- Received Bachelor of Science degree in Information & Computer Science from King Fahd University of Petroleum & Minerals (KFUPM), Dhahran, Saudi Arabia in February 2003.

- Currently working as a System Analyst in Saudi Aramco Information Technology since March 2003.

- Completed the Master of Science degree in Information & Computer Science from KFUPM, as a part time graduate student, in June 2005.